Seria Linux este coordonată de **dr. Gabriel Ciobanu**

Dragoş Acostăchioaie este programator de sistem şi aplicaţii Linux, administrator de reţea la firma BIOSFARM din Iaşi. Predă cursul de „Programarea Calculatoarelor" la Colegiul de Tehnologia Informaţiei din cadrul Universităţii „Al.I. Cuza". Este autorul a numeroase articole despre sistemul de operare Linux, având publicată la editura Polirom lucrarea *Administrarea şi configurarea sistemelor Linux*. Domenii principale de interes: sistemele deschise, programarea orientată-obiect, aplicaţiile client-server şi securitatea sistemelor UNIX.

© 2002 by Editura POLIROM

www.polirom.ro

Editura POLIROM
Iaşi, B-dul Copou nr. 4, P.O. BOX 266, 6600
Bucureşti, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33, P.O. BOX 1-728, 70700

Descrierea CIP a Bibliotecii Naţionale a României:

ACOSTĂCHIOAIE, DRAGOŞ
Programare C şi C++ pentru Linux / Dragoş Acostăchioaie;
cuvânt înainte de Conf. dr. Dorel Lucanu; Iaşi; Polirom, 2002.
216 p.; 24 cm

ISBN: 973-681-112-3

I. Lucanu, Dorel (pref.)

004.43 C
004.43 C++

Printed in ROMANIA

Dragoş Acostăchioaie

PROGRAMARE C ŞI C++ PENTRU LINUX

Cuvânt înainte de Conf. dr. Dorel Lucanu

130252
B.C.U. - IASI

POLIROM
2002

BIBL. CENTR. UNIV.
„M. EMINESCU" IAȘI
INFORMATICĂ

CUPRINS

Cuvânt înainte 9
Prefață 11
Ce este Linux? 14

Partea întâi

PROGRAMAREA ÎN LIMBAJUL C

Capitolul 1
Scurt istoric 19

Capitolul 2
Noțiunile de bază ale limbajului C 20
2.1. Variabile 21
2.2. Constante 23
2.3. Operatori 24
2.4. Funcții 28
2.5. Tablouri 30
2.6. Enumerații 30
2.7. Comentarii 31

Capitolul 3
Utilizarea datelor 32
3.1. Domeniul variabilelor 32
3.2. Conversii de tip 33

Capitolul 4
Controlul fluxului 35
4.1. Instrucțiuni 35
4.2. *if-else* 35
4.3. *switch* 36
4.4. *while* 37
4.5. *for* 38
4.6. *do-while* 39
4.7. *break* și *continue* 39
4.8. *goto* 40

Capitolul 5
Pointeri 41
5.1. Tablouri şi pointeri 43
5.2. Tablouri de pointeri 44
5.3. Pointeri la funcţii 47
Capitolul 6
Structurarea programelor: funcţiile 49
6.1. Recursivitatea 49
6.2. Funcţia *main()* 50
Capitolul 7
Structuri 51
7.1. Structuri cu autoreferire 53
7.2. *typedef* 54
Capitolul 8
Preprocesorul C 55
Capitolul 9
Bibliotecile standard C 57
Capitolul 10
Probleme propuse 63

Partea a doua

PROGRAMAREA ÎN LIMBAJUL C++

Capitolul 1
Scurt istoric 67
Capitolul 2
Noţiunile de bază ale programării orientate-obiect 69
2.1. Premisele limbajelor orientate-obiect 69
2.2. Concepte fundamentale 70
Capitolul 3
Clase 72
3.1. Declararea claselor 72
3.2. Membrii unei clase 73
3.3. Crearea şi distrugerea obiectelor 74
3.4. Conceptul de moştenire 77
Capitolul 4
Programare avansată utilizând clase 80
4.1. Controlul accesului la clase 80
4.2. Funcţii şi clase prietene 82
4.3. Cuvântul-cheie *this* 83
4.4. Redefinirea operatorilor 84
4.5. Moştenirea multiplă 89
4.6. Conversii de tip definite de programator 91
4.7. Constructorul de copiere 93
4.8. Clase abstracte 93
4.9. Membri statici ai unei clase 94
Capitolul 5
Fluxuri 96
5.1. Introducere 96
5.2. Obiecte standard 97
5.3. Redirectări 97
5.4. *cin* 98
5.5. *cout* 100
5.6. Operaţii de intrare/ieşire cu fişiere 102
Capitolul 6
Tratarea excepţiilor 103
Capitolul 7
Template-uri 106
Capitolul 8
Proiectarea şi dezvoltarea de aplicaţii orientate-obiect 121
Capitolul 9
Probleme propuse 125

Partea a treia

UNELTE DE PROGRAMARE PENTRU LINUX

Capitolul 1
Editoare 129
1.1. Editorul *Emacs* 129
1.2. Alte editoare: *mcedit, joe* şi *vi* 132
Capitolul 2
Compilatoare 135
2.1. Compilarea programelor C: *GCC* 136
2.2. Compilarea programelor C++: *G++* 138
2.3. Crearea de biblioteci: *ar* 139
Capitolul 3
Instrumente de programare 140
3.1. Depanarea programelor: *GDB* 140
3.2. Utilitarul *make* 143
3.3. *autoconf, automake* şi *libtool* 148
3.4. Utilitarele *diff, patch* şi *diffutils* 166

3.5. Sistemul de control al versiunilor CVS 170
3.6. Alte programe utile 174
3.7. Documentarea programelor: *DOC++* şi *code2html* 178
3.8. Verificarea programelor: *splint* 181

Capitolul 4
Medii integrate de dezvoltare 183
4.1. Depanatorul vizual *DDD* 183
4.2. Mediul integrat *KDevelop* 184
4.3. Programul *Glade* 186

Capitolul 5
Programarea *open source* 187
5.1. Ce este *open source*? 187
5.2. Managementul Web al proiectelor software – `sourceforge.net` 190

Anexe

Anexa 1
Funcţiile standard ANSI C 195

Anexa 2
Funcţiile standard POSIX 199

Anexa 3
Licenţa Publică GNU 204

Bibliografie 211
Glosar de termeni 213

CUVÂNT ÎNAINTE

Mişcarea *free software* este mai mult o filosofie de viaţă decât una de preţ şi se referă la realizarea de programe-calculator gratuite. Această filosofie se referă la libertatea de a rula un program pentru orice scop dorit, la libertatea de a studia modul de funcţionare al programului, la libertatea de a redistribui copii prietenilor şi colegilor, la libertatea de a aduce îmbunătăţiri şi de a le oferi public în folosul comunităţii. Scopul acestei mişcări este unul social, în sensul că propune un anumit comportament atât producătorilor de programe-calculator, cât şi utilizatorilor acestor programe. De multe ori *free software* este confundat cu termenul (mişcarea) *open source*, care s-ar putea traduce prin *cod-sursă deschis*. Diferenţa este una de ordin practic: *open source* este o metodologie de elaborare a programelor-calculator pentru care *free software* este doar o subpolitică. Cele două mişcări formează împreună *comunitatea free software*.

Unul dintre fenomenele majore ale mişcării *free software*, dacă nu cumva cel mai important dintre ele, este proiectul GNU. Acesta a fost iniţiat de Richard Stallman în 1983 având scopul de a crea un *free software* compatibil cu sistemul Unix (Gnu's Not Unix!). Inevitabil, acest software trebuia să includă şi un sistem de operare. La început au fost dezvoltate câteva utilitare: editorul Emacs, un compilator C, un interpretor de comenzi etc. În anii '90, Linus Torvalds a dezvoltat un nucleu de sistem de operare pe care l-a numit *Linux*. Acest nucleu a fos integrat cu celelalte componente şi a format un sistem de operare complet, cunoscut ca sistem *GNU bazat pe Linux* (*linux-based GNU system*). Astăzi, când spunem sistem de operare Linux, înţelegem de fapt un sistem GNU bazat pe Linux.

Prezenta carte este o invitaţie de a intra în comunitatea *free software*. Pentru a deveni un membru deplin al acestei comunităţi, ai nevoie să cunoşti instrumentele create şi utilizate de către ea. Autorul s-a oprit la limbajele C şi C++, cele mai utilizate în momentul de faţă, şi la principalele programe-utilitare strict necesare în dezvoltarea programelor. Accentul este pus pe modul în care sunt utilizate aceste instrumente, fiind prezentate multe exemple tipice. Sunt convins că această carte îi va face pe mulţi dintre (viitori) programatori să se simtă mai liberi în timpul exercitării profesiei.

Dorel Lucanu
Facultatea de Informatică
a Universităţii „Al.I. Cuza" din Iaşi

PREFAȚĂ

Avându-şi rădăcinile în anii de început ai descoperirilor ştiinţei calculatoarelor moderne, limbajul C, sistemul de operare UNIX şi Internetul se întrepătrund, nici unul dintre ele nu ar putea fiinţa fără celelalte două.

Apariţia sistemului Linux a constituit un nou imbold în cadrul evoluţiei UNIX-ului şi a curentului *open source*. Totul se bazează pe un fundament numit *cooperare*, care în definitiv se bazează pe oameni. Cercetători ca Thompson, Brian Kerninghan, Dennis Ritchie şi Bjarne Stroustrup, iar mai târziu Richard Stallman şi Linus Torvalds au înţeles că rezultatele ştiinţei trebuie împărtăşite cu ceilalţi, cu membrii comunităţii.

Fenomenul *open source* nu a condus doar la elaborarea unui număr uriaş de programe de calitate, liber disponibile prin intermediul Internetului. Acest curent a schimbat mentalitatea oamenilor, făcându-i să gândească nu egoist, ci în folosul unei comunităţi.

Concomitent cu multiplele posibilităţi de utilizare ca server, sistem personal sau la birou, Linux este o platformă ideală pentru dezvoltarea de programe. Instrumentele de programare GNU îşi au originile în anii '80, ele nefiind inventate la comandă, ci izvorând din nevoile programatorilor de a-şi uşura munca.

Puterea instrumentelor GNU pentru mediile UNIX rezidă în capacitatea extraordinară a acestora de a interacţiona. Ideea nu este de a construi monoliţi complecşi, greu de întreţinut, de dezvoltat, care rezolvă toate problemele, ci multe componente mici, făcute să comunice între ele într-un mod natural, standardizat. Un mediu integrat, spre exemplu, nu implementează un depanator propriu, ci utilizează programul GDB, permiţând manipularea acestuia prin comenzi vizuale, dar oferind totodată accesul la linia de comandă. La fel stau lucrurile şi în cazul suportului pentru CVS şi aşa mai departe.

În lucrarea de faţă sunt trecute în revistă noţiunile de bază ale programării C şi C++ şi, totodată, particularităţile acestea pe sistemele Linux. De asemenea, sunt prezentate principalele instrumente şi medii de programare în sistemele de operare Linux şi UNIX, în special pentru limbajele C şi C++.

Cartea conţine şi o prezentare a fenomenului *open source*, îndemnând cititorul să utilizeze software liber şi chiar să contribuie activ la dezvoltarea acestui tip de software.

Aşteptăm reacţiile, remarcele şi eventualele observaţii sau corecţii pe adresa `dragos@biosfarm.ro`. De asemenea, îi invităm pe cititori să viziteze pagina dedicată acestui volum, `http://www.biosfarm.ro/~dragos/prg`.

Structură generală

Lucrarea este structurată în părţi: în prima sunt cuprinse noţiunile de bază ale programării în limbajul C, în timp ce partea a doua tratează elementele esenţiale ale programării în limbajul C++. Ambele segmente conţin numeroase exemple şi propun o serie de probleme. Partea a treia prezintă principalele instrumente de programare din mediile Linux şi UNIX, cum ar fi compilatoarele – `GCC`, `G++`, instrumentele pentru depanarea programelor – GDB, cele pentru automatizarea compilării şi realizarea de programe complexe – `make`, `autoconf`, `automake`, `libtool` sau cele destinate lucrului în echipă – CVS. Sunt prezentate totodată o serie de medii integrate de dezvoltare existente pe platformele menţionate.

Lucrarea include şi o listă de referinţe bibliografice necesare pentru însuşirea temeinică şi aprofundarea limbajelor C şi C++.

Cui se adresează această carte

Cartea se adresează atât studenţilor şi elevilor care urmează cursuri de programare, cât şi specialiştilor în programare, care doresc fie să înveţe sau să utilizeze platforma Linux, fie să îşi porteze programele pe aceasta.

Convenţii utilizate în această carte

Bold-ul este utilizat pentru opţiuni, cuvinte-cheie etc., având rolul de a le evidenţia.

Italic-ul este utilizat pentru nume de variabile, constante, cuvinte-cheie etc., care vor trebui în general, să fie înlocuite cu valoarea lor reală de către utilizator.

`Courier`-ul este folosit pentru nume de comenzi, de fişiere, de utilizatori sau grupuri, precum şi pentru texte care trebuie tastate efectiv de către utilizator. Este de asemenea utilizat pentru conţinutul unui fişier sau ieşirea unor comenzi, în cadrul exemplelor.

| semnifică SAU logic. Astfel, `optiune1 | optiune2` înseamnă „fie `optiune1`, fie `optiune2`".

[...] semnifică un text opţional, deci care nu este obligatoriu de folosit.

. . . semnifică o porţiune de text care a fost omisă pentru a nu îngreuna lizibilitatea sau pentru a reduce din spaţiul utilizat.

este promptul implicit al interpretorului de comenzi pentru utilizatorul *root*.

Cum a fost scris acest material

Această lucrare a fost realizată în HTML 4, utilizând doar programe *open source*: Red Hat Linux 7.3, GNU Emacs, XFig, Mozilla, Konqueror, GIMP.

Programele au fost compilate cu GCC şi EGCS.

Mulţumiri

Doresc să îmi exprim recunoştinţa pentru ajutorul acordat în corectarea şi definitivarea acestei cărţi *Conf. dr. prof. Dorel Lucanu*, precum şi colaboratorilor – în acelaşi timp prietenilor – *Sabin Buraga* şi *Mihaela Brut*. Mulţumesc de asemenea primului meu dascăl de C, *Conf. dr. Claudia Botez*.

Autorul

Iaşi, septembrie 2002

CE ESTE LINUX?

Linux este un sistem de operare gratuit (*open source*), compatibil cu UNIX. UNIX este un sistem multitasking şi multiuser, în care utilizatorii au acces la resursele calculatorului de la diferite terminale plasate local sau la distanţă, putând executa în mod concurent un număr nedeterminat de programe. Linux respectă standardele POSIX, suportă toată gama de aplicaţii GNU şi posedă o interfaţă grafică X Windows System.

Linux a apărut iniţial ca un proiect al lui Linus Torvalds, student pe atunci la Universitatea din Helsinki. Prima versiune utilizabilă a fost lansată pe 5 octombrie 1991. Din acel moment, numărul celor care lucrează la Linux, precum şi numărul de utilizatori, a crescut în mod impresionant. În momentul faţă se estimează că există circa 18 milioane de utilizatori de Linux (pentru detalii, a se vedea situl `http://counter.li.org`).

Spre deosebire de alte sisteme de operare, nici o firmă nu este proprietara sistemului Linux. Mai mult decât atât, el este protejat de Licenţa Publică GNU (*GNU* este o fundaţie care apără interesele autorilor de programe gratuite – pentru informaţii suplimentare, a se vizita situl `http://www.gnu.org`), care stipulează faptul că se poate copia şi utiliza gratuit codul programelor, cu condiţia de a se permite şi altora să facă acelaşi lucru. Textul complet al acestei licenţe se găseşte în Anexa 3.

Sistemul Linux propriu-zis este alcătuit dintr-un nucleu (*kernel*) şi un număr uriaş de programe şi utilitare. Este un sistem de operare complet, care conţine practic orice tip de aplicaţie, putând astfel juca o varietate de roluri, printre care:

- **Server.** Sistemul Linux conţine un suport excepţional pentru reţea, putând oferi atât serviciile specifice Internetului, cum ar fi Web, FTP, poştă electronică etc., precum şi alte servicii de reţea, cum ar fi server de imprimante, server de fişiere (NFS) ş.a.m.d. De asemenea, Linux poate substitui un server Windows NT sau Netware. Stabilitatea, fiabilitatea şi securitatea sa îi conferă posibilitatea de a fi utilizat pe servere *high-end* şi în medii critice.
- **Sistem personal.** Linux poate fi utilizat la redactarea de texte, tipărirea la imprimantă, conectarea şi navigarea pe Internet, citirea corespondenţei. În plus, este o platformă multimedia perfectă pentru jocuri, pentru vizionarea de filme sau audiţia de piese muzicale.
- **Sistem utilizat la serviciu.** Sistemul Linux conţine o suită completă de programe pentru birou, cum ar fi programe de calcul tabelar, redactare de texte, realizarea de prezentări etc. Instrumentele pentru acces la Internet reprezintă şi ele un punct forte al acestui sistem.
- **Sistem educaţional.** Atuul principal este costul practic nul al sistemului. Instalarea de sisteme proprietare pe multe calculatoare poate fi extrem de costisitoare. Linux include aplicaţii, compilatoare şi medii integrate de dezvoltare a căror calitate este în general superioară celei a programelor comerciale.

Un sistem Linux, alcătuit dintr-un program de instalare, un nucleu şi aplicaţii gata compilate se numeşte *distribuţie Linux*. Numeroase companii şi organizaţii au realizat asemenea distribuţii. Acestea sunt disponibile gratuit pe Internet, dar pot fi achiziţionate şi contra cost ca pachete formate din CD-ROM-uri şi manuale de utilizare. Distribuţiile Linux cele mai răspândite sunt *Red Hat*, *Mandrake*, *SuSE*, *Caldera* şi *Slackware*. Distribuţia *Red Hat* poate fi descărcată de la adresa `http://www.redhat.com` sau *mirrors* (oglindiri ale unui sit Internet), iar *Mandrake*, de la adresa `http://www.mandrake.com`.

Contrar opiniei generale, suportul tehnic pentru programele *open source* are de multe ori o calitate superioară celui oferit pentru aplicaţiile comerciale. Prin formularea unei întrebări sau unei cereri de ajutor pe o listă de discuţie, se poate obţine un răspuns la orice problemă, în câteva ore. De asemenea, există numeroase companii, îndeosebi cele care comercializează distribuţii, ce oferă consultanţă şi suport tehnic profesional, contra cost.

Nucleul Linux a fost adaptat (*portat*) pe foarte multe platforme, începând cu Intel şi continuând cu SUN Sparc, PowerPC, Motorola 68000 (Atari şi Amiga), MIPS (SGI), DEC Alpha, ARM etc.

În ultimii ani, Linux a cunoscut o largă răspândire şi în România. Fiind gratuit, nu există posibilitatea de a intra în conflict cu legea dreptului de autor, ceea ce va conduce cu siguranţă la utilizarea sa şi în cadrul firmelor, precum şi în mediile cu multe calculatoare. În majoritatea centrelor universitare din ţară, Linux este extrem de popular, multe materii folosindu-l ca platformă de referinţă (Sisteme de operare, Programare ş.a.m.d.). Cei mai mulţi furnizori de servicii Internet folosesc Linux datorită capabilităţilor sale excelente de reţea, flexibilităţii în configurare şi stabilităţii sale. De asemenea, profesioniştii ştiinţei calculatoarelor îl utilizează pentru instrumentele puternice de programare, procesare de texte etc. de care dispune.

În seria Linux, din care face parte şi prezentul volum, au apărut şi vor apărea lucrări de referinţă privind programarea, utilizarea şi administrarea în cadrul acestui sistem de operare.

Partea întâi

PROGRAMAREA ÎN LIMBAJUL C

Capitolul 1

SCURT ISTORIC

Fă aşa!
(Jean Luc Picard)

În anul 1970, doi programatori, Brian Kerningham şi Dennis Ritchie, de la AT&T (*Bell Laboratories*), au creat limbajul *C*. Principalul scop pentru care a fost realizat acest limbaj a fost rescrierea sistemului de operare UNIX, pentru a-l face portabil pe toate platformele existente. Din acest motiv, majoritatea programelor pentru UNIX sunt scrise tot în C. Limbajul C este un descendent al limbajelor *B*, *MACRO11* şi *BCPL*. Prima versiune a limbajului C a rulat pe un calculator de tip *PDP-11*.

Marele avantaj al limbajului C este acela de a fi extrem de flexibil şi de a permite programarea atât la nivel înalt, cât şi la nivel scăzut. C-ul este un limbaj procedural, ceea ce face ca un program scris în C să înceapă de obicei cu definirea structurilor de date şi să continue cu definirea funcţiilor pentru lucrul cu aceste structuri.

C este un limbaj de programare cu scop general şi nu este dependent în vreun fel de hardware sau de maşina pe care se utilizează. Un program în C ar trebui să se poată compila şi executa fără nici un fel de modificări pe orice calculator, atât timp cât se respectă standardele universal acceptate (ANSI C).

Primul manual de programare C a fost realizat în 1978 de către Brian Kerningham şi Dennis Ritchie şi denumit *The C Programming Language*, cunoscut şi sub numele de „*cartea albă*" sau „*K&R*."

În anul 1989, limbajul C a fost standardizat oficial de către comitetul ANSI X3J11.

Capitolul 2

NOȚIUNILE DE BAZĂ ALE LIMBAJULUI C

Nu este corect. Nu este nici măcar greșit.
(Wolfgang Pauli)

C este un limbaj cu un nivel relativ inferior. Astfel, acesta nu posedă operații pentru lucrul cu variabile compuse, cum ar fi șiruri de caractere, liste sau tablouri. De asemenea, limbajul nu prevede instrucțiuni de alocare a memoriei, nu conține nici instrucțiuni de intrare/ieșire, și nici de lucru cu fișiere. Toate aceste mecanisme, considerate a fi de nivel înalt, trebuie apelate prin intermediul funcțiilor conținute în biblioteca standard C sau în alte biblioteci. Această caracteristică prezintă avantajul că executabilele (codul-obiect) generate au dimensiune redusă, funcțiile de bibliotecă putând fi legate dinamic de acestea.

Programele C care utilizează doar bibliotecile standard pot fi compilate și executate pe orice combinație mașină-compilator, fără nici o modificare, în situația în care nu utilizează facilități specifice unui anumit compilator. Conversia unui program-sursă de pe o mașină pe alta se numește *portare*.

Pașii care trebuie urmați în vederea creării unui fișier executabil sunt:

1. Se creează fișierul-sursă, cu ajutorul unui editor de texte.
2. Se compilează fișierul-sursă, în vederea generării fișierului-obiect.
3. Se link-editează fișierul-obiect, generându-se fișierul executabil.

Dacă se utilizează un mediu integrat, acesta include un editor de texte. În caz contrar, poate fi folosit orice editor de texte, cum ar fi `vi`, `emacs`, `joe` etc. Editorul de texte nu trebuie totuși să introducă caractere speciale sau alte caractere de formatare, ci să genereze doar text ASCII. Fișierele create cu ajutorul editorului se numesc *fișiere-sursă*.

Pentru a urma tradiția manualelor de C, vom prezenta în continuare cel mai scurt program posibil, care va afișa pe ecran mesajul „*hello, world*":

```
main()
{
 printf("hello, world\n");
}
```

Presupunând că fișierul-sursă în care a fost salvat programul de mai sus se numește `hello.c`, compilarea acestuia se poate face prin comanda:


```
gcc -o hello hello.c
```

Fișierul executabil rezultat se va numi `hello` și poate fi lansat în execuție cu ajutorul comenzii:

```
./hello
```

2.1. Variabile

Variabilele sunt unitățile de informație care se pot modifica de-a lungul execuției programului. Ele au o durată de viață bine determinată, în funcție de locul și modul de declarare (vezi *infra*).

Numele variabilelor se specifică prin intermediul identificatorilor și sunt alcătuite din litere și cifre; primul caracter trebuie să fie o literă. Poate fi utilizat și caracterul „_", care joacă tot rol de literă. Literele mari și mici sunt considerate caractere distincte; în mod convențional, literele mici sunt folosite pentru variabile, iar cele mari, pentru constante simbolice. Numărul de caractere din cadrul numelui este nelimitat. O serie de cuvinte-cheie nu pot fi fi utilizate ca nume de variabile:

auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	unsigned
union	void	volatile	while			

Există doar următoarele tipuri de date de bază în C:

char	un caracter
int	un număr întreg. Numărul de biți în care este reprezentată valoarea depinde de tipul calculatorului gazdă (vezi *infra*)
float	număr flotant în simplă precizie
double	număr flotant în dublă precizie

În plus, există doi calificatori care pot fi aplicați tipului `int`: `short` și `long`, referindu-se la diferite mărimi de întregi. De asemenea, tipurilor `char` și `int` li se poate aplica calificatorul `unsigned` (fără semn). Cuvântul `int` poate fi omis în aceste cazuri.

Pe sistemele Linux i386 (compatibile IBM PC):

Tip	Lungime	Plajă valori
char	8 biţi	cu semn: -128 – 127, fără semn: 0 – 255
int	32 biţi	cu semn: -2147483648 – 2147483647, fără semn: 0 – 4294967295
short	16 biţi	cu semn: -32768 – 32767, fără semn: 0 – 65536
long	32 biţi	cu semn: -2147483648 – 2147483647, fără semn: 0 – 4294967295
float	32 biţi	$3.4*10^{-38} – 3.4*10^{38}$
double	64 biţi	$1.7*10^{-308} – 1.7*10^{308}$

Pentru a afla valoarea minimă, respectiv cea maximă pe care o poate memora o variabilă de tip numeric, trebuie consultat fişierul `/usr/include/ limits.h`, care conţine definiţii de genul `CHAR_BIT` – numărul de biţi dintr-un `char`, `CHAR_MAX` – valoarea maximă a unui `char` ş.a.m.d.

Toate variabilele trebuie declarate înainte de a fi utilizate. O declaraţie menţionează un tip şi este urmată de una sau mai multe variabile de acel tip, specificate prin numele acestora şi separate prin virgulă:

```
int x, y, z;
char s[1000];
```

Variabilele pot fi, de asemenea, iniţializate în momentul declarării lor, cu o valoare iniţială, precizând după numele acestora semnul egal „=" şi o constantă:

```
int i = 0;
char s[20] = "Salut";
```

De remarcat este faptul că la declararea şirului de caractere `s` se alocă în mod automat 20 de caractere, chiar dacă lungimea valorii iniţiale este mai mică decât atât.

Variabilele nu sunt iniţializate implicit de către compilator. Variabilele neiniţializate explicit au valoare nedefinită (engl. *garbage*), cu alte cuvinte programatorul nu trebuie să se bazeze pe o anumită valoare a acestora.

Variabile de tip *const*

Modificatorul de acces `const` este o promisiune a programatorului pentru compilator că valoarea unei variabile nu va fi modificată după iniţializarea sa. Dacă aceasta nu este iniţializată, compilatorul o va iniţializa pe zero. În acest sens, este important de remarcat că declaraţia de genul `const char *str = "hello"` este un pointer către o informaţie care nu poate fi modificată. Pointerul în sine poate fi modificat (vezi capitolul 5).

Variabilele de acest tip sunt utile în general pentru declararea parametrilor funcţiilor (vezi paragraful 2.4). Spre exemplu, o funcţie care contorizează numărul de caractere dintr-un şir nu trebuie să modifice conţinutul acestuia:

```
int lungime_sir(const char *sir)
{
    int contor = 0;

    while(*sir++)
       contor++;

    return contor;
}
```

Variabile de tip *register*

Cuvântul-cheie `register` precizează compilatorului că variabila precizată este des utilizată şi ar trebui memorată într-una dintre registrele procesorului. Pe compilatoarele moderne, nu este necesar să fie utilizat. Multe dintre aceste compilatoare ignoră chiar această directivă.

2.2. Constante

O constantă numerică desemnează o valoare numerică fixată.

Sufixul „L" sau „l" ataşat unei constante numerice forţează reprezentarea sa ca fiind de tip `long`. Sufixul „U" sau „u" forţează reprezentarea `unsigned`, iar dacă depăşeşte dimensiunea maximă a acestuia, va fi automat de tip `unsigned long`. Cele două sufixe pot fi utilizate şi împreună.

Toate constantele în virgulă flotantă sunt prin definiţie de tip `double`. Forţarea la tipul `float` poate fi făcută prin adăugarea sufixului „F" sau „f".

Pentru constantele numerice sunt permise şi notatiile de genul `123.456e-7` sau `0.123E3`. Există de asemenea o notaţie specială care permite exprimarea de valori octale şi hexazecimale: „0" la începutul unei constante înseamnă că aceasta este exprimată în octal; prefixul `0x` înseamnă că valoarea este exprimată în hexa.

O constantă de tip caracter este un singur caracter, scris între apostroafe ' '. Valoarea este corespondentul numeric al caracterului în setul de caractere al maşinii. De exemplu, în setul de caractere ASCII, caracterul „0" are valoarea 48. Constantele de tip caracter participă în operaţiile numerice ca orice alte valori numerice, de exemplu în cadrul operaţiei de comparaţie cu alte caractere.

Anumite caractere standard pot fi reprezentate cu ajutorul *secvenţelor escape*, cum ar fi `\n` (linie nouă), `\t` (tab), `\\` (backspace), `\0...` (corespondentul valorii specificate în octal) sau `\0x...` (corespondentul valorii specificate în hexa).

O constantă-şir de caractere este o secvenţă compusă din zero sau mai multe caractere, între ghilimele duble ", ca `"Hello, world!"` sau `""` (un şir nul). Secvenţele escape descrise mai sus sunt valabile şi în acest caz. Utilă poate fi secvenţa `\"`, care înlocuieşte caracterul ghilimele duble (imposibil de utilizat ca atare).

Practic, un şir de caractere este un tablou ale cărui elemente sunt caractere de tip `char`. Compilatorul plasează în mod automat un caracter nul, `\0` (având practic valoarea numerică zero), la sfârşitul şirului, caracter denumit *terminator nul*. Funcţia următoare, `lungime_sir(s)`, returnează lungimea unui şir de caractere `s`, primit ca argument, exclusiv terminatorul `\0`:

```
int lungime_sir(char *s)
{
    int i = 0;

    while(s[i] != '\0')
       i++;

    return i;
}
```

La manipulările de şiruri de caractere trebuie avut în vedere că este necesar a fi alocată suficientă memorie pentru a memora întregul şir, inclusiv acest caracter de terminare.

2.3. Operatori

Operatorii specifică operaţiile care se fac cu variabile şi constante, conform tipurilor acestora.

2.3.1. Operatorul de atribuire

Operatorul de atribuire este semnul egal „=". Valoarea din dreapta semnului egal este atribuită variabilei din stânga. Într-o instrucţiune se pot face mai multe atribuiri:

```
rez = x = y;
```


În acest caz, ordinea evaluării este de la dreapta la stânga. Astfel, `y` va fi atribuit lui `x`, care la rândul său va fi atribuit lui `rez`.

2.3.2. Operatori aritmetici

Operatorii aritmetici binari sunt „+", „-", „*", „/" şi operatorul modulo, „%". Există operatorul unar „-", dar nu există operatorul unar „+". De notat este faptul că împărţirea întregilor trunchiază orice parte fracţionară. Operatorul „%" nu poate fi aplicat la `float` sau `double`.

2.3.3. Operatori relaţionali

Aceşti operatori permit compararea a două valori. Dacă rezultatul comparaţiei este fals, valoarea rezultată este 0 (zero), respectiv 1 în caz contrar. Operatorii relaţionali sunt: > (mai mare), >= (mai mare sau cel mult egal), < (mai mic), <= (mai mic sau cel mult egal), == (egal), != (inegal).

2.3.4. Operatori logici

Aceştia sunt && şi (ŞI logic), || (SAU logic), numiţi şi conectori logici, şi „!" (negaţie logică). Expresiile care conţin operatori logici sunt evaluate de la stânga la dreapta, iar evaluarea se opreşte în momentul în care se cunoaşte rezultatul (cu alte cuvinte, dacă rezultatul nu mai poate fi modificat de operaţiile rămase, acestea sunt abandonate).

În C nu există un tip de date corespunzător tipului `bool`, acesta existând în schimb în limbajul C++. În cazul operaţiilor logice, o valoare nenulă (în mod convenţional 1) este considerată ca având valoarea de adevăr *adevărat*, iar valoarea 0 desemnează *fals*.

Operatorul unar de negaţie „!" converteşte un operand non-zero sau adevărat în zero şi un operand egal cu zero sau fals în 1. Spre exemplu, expresia:

```
if(!ok)
```

este echivalentă cu:

```
if(ok == 0)
```

2.3.5. Operatorii de incrementare şi decrementare

Operatorul de incrementare ++ adună 1 la operandul său, iar -- scade 1 din operand. Aceşti doi operatori pot fi utilizaţi atât ca prefix (înaintea variabilei, *e.g.* ++i), cât şi ca sufix (după variabilă, *e.g.* i++). Efectul asupra operandului este acelaşi, dar folosirea cu prefix modifică variabila înainte de a-i folosi valoarea, în timp ce utilizarea ca sufix modifică variabila după ce valoarea sa a fost folosită în cadrul expresiei.

Folosind această facilitate, putem simplifica unele construcţii, cum ar fi funcţia de calcul a lungimii unui şir de caractere, descrisă în paragraful 2.2:

```
int lungime_sir(char *s)
{
    int i = 0;

    while(s[i++] != '\0');

    return i;
}
```

2.3.6. Operatori logici pe biţi

Operatorii logici pe biţi sunt următorii: „&" (ŞI bit cu bit), „|" (SAU bit cu bit), „^" (SAU EXCLUSIV bit cu bit), << (deplasarea biţilor spre stânga), >> (deplasarea biţilor spre dreapta), şi „~" (complement faţă de 1). Aceşti operatori nu se pot utiliza pentru valori de tip `float` sau `double`.

Operatorul ŞI „&" este utilizat în general pentru a masca anumiţi biţi ai unui număr. Mascarea biţilor reprezintă selectarea unor anumiţi biţi dintr-un octet. Pentru aceasta, se aplică operaţia ŞI bit cu bit cu un număr având setaţi biţii respectivi pe 1. De exemplu, pentru a verifica bitul cel mai puţin semnificativ a variabilei `flags`, efectuăm `flags & 1`.

Operatorul SAU „|" este folosit pentru a seta anumiţi biţi pe 1:

```
x = x | 12;
```

setează pe 1 biţii 2 şi 3 din x, deoarece $12_{10} = 1100_2$.

Operaţiile de deplasare << şi >> realizează deplasări la stânga, respectiv la dreapta pentru operandul din stânga lor, cu numărul de poziţii dat de operandul din dreapta lor. Deplasarea la stânga va provoca umplerea locurilor libere din dreapta cu zero. Deplasarea la dreapta va umple biţii liberi din stânga cu zero. Spre exemplu: `4 << 1` = 12 şi `12 >> 2` = 2, deoarece $2_{10} = 10_2$, $4_{10} = 100_2$ şi $12_{10} = 1100_2$.

Operatorul unar „~" furnizează complementul faţă de 1 al unui întreg, adică converteşte fiecare bit de 1 în 0 şi viceversa.

2.3.7. Operatori compuşi de asignare

Majoritatea operatorilor binari permit ataşarea unui operator de asignare de forma *operator*=: +=, -=, *=, /=, %=, <<=, >>=, &=, |= şi ^=. Astfel, expresii de genul `i = i + 1` pot fi simplificate prin `i += 1`.

Funcţia de mai jos calculează numărul de biţi setaţi pe 1 dintr-un număr întreg:

```
int bitcount(unsigned n)
{
    int i;

    for(i = 0; n != 0; n >>= 1)
       if(n & 1)
          i++;

    return i;
}
```

2.3.8. Ponderea operatorilor şi ordinea lor de evaluare

În tabelul de mai jos, operatorii aflaţi pe aceeaşi linie au pondere egală; liniile tabelului sunt în ordinea descrescătoare a ponderilor:

Operator	Modul de evaluare
() [] -> .	de la stânga la dreapta
! ~ ++ -- - (*tip*) * & sizeof (operatori unari)	de la dreapta la stânga
* / %	de la stânga la dreapta
+ -	de la stânga la dreapta
<< >>	de la stânga la dreapta
< =&t;= > >=	de la stânga la dreapta
== !=	de la stânga la dreapta
&	de la stânga la dreapta
^	de la stânga la dreapta
|	de la stânga la dreapta
&&	de la stânga la dreapta
||	de la stânga la dreapta
? :	de la dreapta la stânga
= += -= etc.	de la dreapta la stânga

2.4. Funcţii

Funcţiile reprezintă un set de instrucţiuni grupate într-un bloc de program. Ele permit împărţirea programelor în mai multe subrutine mai mici. Funcţiile ascund deseori detaliile de implementare ale unor operaţii specifice, pe care programatorul nu trebuie neapărat să le cunoască. Astfel, funcţiile uşurează munca de scriere a codului, facilitează lucrul în echipă şi introduc ideea de reutilizare a codului (această idee permite programatorilor să utilizeze munca altora, fără a începe totul de la capăt). Funcţiile pot fi localizate fizic şi în fişiere diferite, reducând în acest mod dimensiunile codului-sursă. De asemenea, funcţiile pot să rezide în biblioteci (cum ar fi cele standard).

Funcţiile pot primi un număr nelimitat de argumente, pentru fiecare în parte trebuind declarat tipul şi numele. Numele este necesar pentru a putea accesa respectivul argument din cadrul funcţiei. Argumentele funcţiei sunt trimise prin valoare, adică funcţia apelată primeşte o copie temporară, locală, a fiecărui argument şi nu adresa sa. Ca urmare, funcţia nu poate afecta valoarea originală a argumentului din funcţia apelantă. Atunci când argumentul este un tablou (*e.g.* un şir de caractere), este trimisă adresa de început a tabloului, deci apelul se face prin referinţă, iar elementele nu sunt copiate. Există de asemenea posibilitatea ca o funcţie să primească un număr variabil de argumente (vezi subcapitolul 6.2).

Funcţiile pot returna o valoare al cărei tip trebuie să preceadă numele funcţiei. Dacă o funcţie nu returnează nici o valoare, tipul declarat trebuie să fie `void`. Instrucţiunea `return` specifică ieşirea din funcţie şi ea poate fi urmată de o valoare, dacă este cazul. Nu este obligatoriu ca funcţiile `void` să apeleze instrucţiunea `return` la sfârşitul acestora.

În exemplul următor vom utiliza funcţia `lungime_sir`, definită mai sus:

```
#include <stdio.h>

int lungime_sir(char *s)
{
    int i = 0;

    while(s[i++] != '\0');

    return i;
}

void main()
{
    char s[1000];

    printf("Introduceti sirul de caractere: ");
    gets(s);
    printf("Sirul are %d caractere.\n", lungime_sir(s));
}
```


Pentru a putea utiliza o funcţie, aceasta trebuie mai întâi declarată. În cadrul declaraţiei, numele argumentelor poate lipsi:

```
int lungime_sir(char *);
```

În mod uzual, declaraţia unei funcţii care este definită în alt fişier se face într-un *fişier-antet* (*header*), care poate fi inclus în orice fişier-sursă care utilizează funcţia respectivă. Fişierul-antet poate conţine şi declaraţii de variabile, constante etc.:

functii.h

```
int lungime_sir(char *);
```

global.c

```
#include "functii.h"

int lungime_sir(char *s)
{
    int i = 0;

    while(s[i++] != '\0');

    return i;
}
```

main.c

```
#include <stdio.h>
/* Linia de mai jos poate fi inlocuita cu
    #include "global.c" */
#include "functii.h"

void main()
{
    char s[1000];

    printf("Introduceti sirul de caractere: ");
    gets(s);
    printf("Sirul are %d caractere.\n", lungime_sir(s));
}
```

Compilarea acestor fişiere se va face cu:

```
gcc -o test global.c main.c
```

2.5. Tablouri

Un tablou multidimensional este practic o matrice. Declararea unui asemenea tablou se face cu:

```
tablou[dimensiune1][dimensiune2]...[dimensiuneN]
```

Un tablou multidimensional se iniţializează printr-o listă de valori iniţiale scrise între acolade. Pentru exemplificare vom implementa o funcţie care returnează numărul de zile ale unei luni:

```
int nr_zile(int luna, int an)
{
    int tabzile[2][12] = {
       { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
       { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
    };

    int bisect = __an % 4 == 0 && __an % 100 != 0 || __an % 400 == 0;

    return tabzile[bisect][__luna - 1];
}
```

2.6. Enumeraţii

Tipul enumerare enum reprezintă o modalitate de a utiliza nume în loc de numere, pentru un tip comun de date:

```
enum figura_geometrica {
    cerc,
    patrat,
    triunghi
};
```

Numelor cerc, patrat şi triunghi le vor fi asignate în mod automat valorile 0, 1 respectiv 2. Numerele corespunzătoare numelor sunt considerate întregi. Pentru a asigna alte valori, se foloseşte o construcţie de forma:

```
#include <stdio.h>

enum figura_geometrica {
    cerc = 10, patrat = 20, triunghi = 30
};
```



```
main()
{
    enum figura_geometrica g = triunghi;

    printf("Valoarea lui g = %d\n", g);
}
```

2.7. Comentarii

Comentariile reprezintă anumite observaţii introduse de programator, ele fiind ignorate la compilare.

Începutul unui comentariu este marcat prin /*, iar sfârşitul acestuia prin */. Comentariile pot cuprinde mai multe linii, ca în exemplul de mai jos:

```
/* Comentariu
      care ocupa
      mai multe linii */
```

Pot fi folosite şi comentarii imbricate, însă acestea nu sunt permise de către unele compilatoare.

Capitolul 3

UTILIZAREA DATELOR

Nu-ţi fă probleme, mă voi gândi la o soluţie.
(Indiana Jones)

Un *bloc local* este o porţiune de program C încadrată de paranteza deschisă „{" şi de paranteza închisă „}". O funcţie C conţine aceste două paranteze, aşadar reprezintă şi ea un bloc local.

Un bloc local poate apărea oriunde în cadrul programului. Variabilele declarate în interiorul unui bloc sunt vizibile doar în cadrul acestuia. Altfel spus, viaţa unei variabile începe o dată cu declararea acesteia şi se încheie o dată cu sfârşitul blocului. Variabilele cu aceleaşi nume declarate într-un bloc local au precedenţă faţă de variabilele declarate în afara blocului.

3.1. Domeniul variabilelor

Unul dintre punctele forte ale limbajului C este flexibilitatea sa în definirea stocării datelor. Sunt două aspecte care pot fi controlate în C: domeniul şi timpul de viaţă a variabilelor. *Domeniul* unei variabile se referă la locurile în care variabila poate fi accesată. *Timpul de viaţă* se referă la momentul în care variabila poate fi accesată.

Există trei domenii ale variabilelor:

- `extern` – implicit pentru variabilele declarate în afara oricărei funcţii. Domeniul variabilei declarate `extern` este întregul program;
- `static` – domeniul unei variabile declarate static în afara oricărei funcţii este restul programului în respectivul fişier-sursă. Domeniul unei variabile declarate `static` în interiorul unei funcţii este restul blocului local;
- auto – implicit pentru variabilele declarate în afara unei funcţii. Domeniul este restul blocului local.

Timpul de viaţă al unei variabile `extern` sau `static` durează dinainte ca funcţia `main()` să fie apelată şi se sfîrşeşte la ieşirea din program. Timpul de viaţă al argumentelor funcţiilor durează până la ieşirea din funcţie (apelul instrucţiunii

`return`). Timpul de viaţă al unui variabile alocate dinamic (pointer) începe o dată cu alocarea spaţiului de memorie prin apelul `malloc()` şi se încheie o dată cu apelul funcţiei `free()` sau la încheierea execuţiei programului.

O variabilă globală care trebuie să poată fi accesată din mai multe fişiere trebuie declarată într-un *fişier-antet*. O asemenea variabilă trebuie definită doar într-un singur fişier-sursă. Variabilele nu trebuie definite în fişiere-antet.

Declararea unei variabile desemnează precizarea tipului său către compilator, dar nealocarea de spaţiu de memorie pentru ea. Definirea variabilei înseamnă declararea acesteia şi alocarea de spaţiu de memorie pentru ea:

```
extern int i;   /* declaratie */
int n;          /* definitie */
```

3.2. Conversii de tip

Atunci când într-o expresie apar operanzi de tipuri diferite, aceştia sunt convertiţi în mod automat într-un tip comun.

În cadrul expresiilor aritmetice, conversia operanzilor se face după următoarele reguli:

Orice operand care nu este de tip `int` sau `double` este convertit astfel:

Tip	Se converteşte la tipul
char	int
unsigned char	int
signed char	int
short	int
enum	int
float	double

1. Dacă unul dintre operanzi este de tip `double`, atunci şi celălalt este convertit la `double`.
2. În caz contrar, dacă unul dintre operanzi este de tip `unsigned long`, celălalt este convertit şi el la `unsigned long`.
3. În caz contrar, dacă unul dintre operanzi este de tip `long`, celălalt este convertit la tipul `long`.

4. În caz contrar, dacă unul dintre operanzi este de tip `unsigned`, celălalt este convertit la `unsigned`.
5. În caz contrar, ambii operanzi sunt de tipul `int`.

Modificarea tipului unui operand (*cast*) dintr-o expresie poate fi făcută şi de către programator. Prima utilizare a acestei tehnici este modificarea tipului unui operand dintr-o operaţie aritmetică astfel încât operaţia să fie efectuată corect, atunci când compilatorul nu poate decide corect o conversie de tip.

A doua utilizare este modificarea tipului de pointeri către sau dinspre `void *`, pentru a realiza interfaţa cu funcţiile care au ca argument sau returnează pointeri de tip `void`. De exemplu:

```
struct student *s = (struct student *)malloc(sizeof(struct student));
```

Capitolul 4

CONTROLUL FLUXULUI

Fereşte-te de programatorii
care au şurubelniţe!
(anonim)

Instrucţiunile de control al fluxului dintr-un limbaj specifică ordinea în care se fac calculele.

4.1. Instrucţiuni

În limbajul C, orice instrucţiune se încheie cu „;", caracter considerat terminator de instrucţiuni.

Acoladele, „{" şi „}" sunt folosite pentru a grupa instrucţiuni şi declaraţii, construcţie echivalentă practic cu o singură instrucţiune. După acolada închisă care încheie un bloc nu se pune „;".

4.2. *if-else*

Instrucţiunea `if-else` se utilizează pentru luarea condiţională de decizii. Sintaxa ei este:

```
if(expresie)
    instructiune1
else
    instructiune2
```

unde secţiunea `else` este opţională. Dacă expresia din paranteza de după `if` este adevărată (adică are o valoare nenulă), este executată `instructiune1`. În caz contrar, este executată `instructiune2`:

```
if(i == 1)
    printf("i egal cu 1\n");
else
    return;
```

Deoarece secţiunea `else` este opţională, se poate ajunge la ambiguităţi atunci când avem de-a face cu structuri `if-else` imbricate. În aceste cazuri, este necesară utilizarea acoladelor:

```
if(i > 0)
{
    if(x > y)
      res = x;
}
else
    res = y;
```

Construcţia `if-else` poate fi înlocuită (simplificată) prin utilizarea operatorului ternar „`? :`", în expresia:

```
e1 ? e2 : e3
```

Se evaluează mai întâi expresia `e1`. Dacă aceasta este adevărată, atunci se evaluează expresia `e2` şi aceasta este valoarea expresiei condiţionale. În caz contrar, se evaluează `e3` şi aceasta va fi rezultatul expresiei condiţionale. Exemplul următor calculează maximul dintre două numere:

```
max = (a > b ? a : b);
```

4.3. *switch*

Instrucţiunea `switch` se utilizează pentru a lua decizii multiple, testând dacă o expresie se potriveşte cu una dintr-un număr de valori constante şi executând corespunzător anumite instrucţiuni. Programul următor contorizează tipurile de caractere dintr-un text introdus de la tastatură:

```
main()
{
    int c, nr_spatii = 0, nr_cifre = 0, nr_car = 0;

    while((c = getchar()) != EOF)
      switch(c)
      {
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
```



```
        case '6':
        case '7':
        case '8':
        case '9':
          nr_cifre++;
          break;
        case ' ':
        case '\t':      /* tab */
        case '\n':      /* linie noua */
          nr_spatii++;
          break;
        default:        /* alte caractere decat cele
                           enumerate mai sus */
          nr_car++;
          break;
      }
    printf("Am intalnit %d cifre, %d spatii si %d alte caractere\n",
      nr_cifre,
      nr_spatii,
      nr_car);
}
```

Fiecare caz trebuie să fie „etichetat" cu o constantă întreagă. Dacă un caz se potriveşte cu valoarea expresiei din paranteză, execuţia este transferată la cazul respectiv, după întâlnirea primului `break` trecându-se la execuţia următoarei instrucţiuni de după `switch`. Cazul denumit `default` este executat atunci când nici unul dintre cazuri nu a fost întâlnit. Instrucţiunea `break` provoacă ieşirea din cadrul `switch`, fără a mai verifica şi celelalte cazuri.

4.4. *while*

Sintaxa instrucţiunii `while` este următoarea:

```
while (expresie)
 instructiune
```

Dacă expresia specificată între paranteze este adevărată (nenulă), este executată instrucţiunea. După execuţie expresia este evaluată şi din nou ciclul continuă până când valoarea expresiei devine falsă (nulă). Exemplul următor va decrementa valoarea contorului `i` până când aceasta devine 0:

```
while(i > 0)
    i--;
```

4.5. *for*

Sintaxa instrucţiunii `for` este:

```
for(e1; e2; e3)
    instructiune
```

Expresia *e1* este evaluată o singură dată, la apelul instrucţiunii `for`. Dacă expresia *e2* este adevărată (nenulă), este executată *instrucţiune* şi apoi evaluată expresia *e3*, după care ciclul este reluat până când *e2* devine falsă (nulă). De obicei, *e1* şi *e3* sunt expresii de asignare, iar *e2* o expresie relaţională. Oricare dintre expresii poate fi omisă, dar caracterele „;" trebuie să rămână. Dacă *e2* este omisă, ea este considerată întotdeauna adevărată. Instrucţiunea `break` în cadrul *instrucţiune* provoacă ieşirea imediată din ciclul `for`. Instrucţiunea `for` este echivalentă cu:

```
e1;
while(e2)
    {
    instructiune
    e3;
    }
```

Astfel, instrucţiunea de mai jos decrementează variabila `i` până când aceasta devine 0:

```
for(i = 5; i > 0; i--)
    ...
```

În cadrul instrucţiunii `for` poate fi utilizat un operator special, virgula, „,". Astfel, o succesiune de expresii separate prin virgulă este evaluată de la stânga la dreapta şi tipul şi valoarea rezultatului sunt tipul şi valoarea operandului cel mai din dreapta. Funcţia prezentată în exemplul următor inversează caracterele dintr-un şir:

```
void reverse(char *s)
{
    int c, i, j;

    for(i = 0, j = strlen(s) - 1; i < j; i++, j--)
    {
       c = s[i];
       s[i] = s[j];
       s[j] = c;
    }
}
```


Iată mai jos o implementare pentru funcţia `atoi`, care converteşte un şir de caractere într-un număr întreg:

```
int atoi(char *s)
{
    int i, n, semn = 1;

    /* ignora spatiile de la inceputul sirului */
    for(i = 0; s[i] == ' ' || s[i] == '\t' || s[i] == '\n'; i++)
       ;
    if(s[i] == '+' || s[i] == '-')
       /* verifica semnul */
       semn = (s[i] == '+' ? 1 : -1);
    /* calculeaza partea intreaga */
    for(n = 0; s[i] >= '0' && s[i] <= '9'; i++)
       n = 10 * n + s[i] - '0';

    return semn * n;
}
```

4.6. *do-while*

Instrucţiunea `do-while` este similară cu `while`, dar condiţia de terminare este evaluată la sfârşitul ciclului şi nu la începutul acestuia:

```
do
    instructiune
while(expresie);
```

4.7. *break* şi *continue*

Am întâlnit instrucţiunea `break` în paragrafele precedente. Aşa cum am văzut, aceasta determină încheierea imediată a ciclului (buclei).

Instrucţiunea `continue` are un efect diametral opus lui `break`, în sensul că face să înceapă următoarea iteraţie a ciclului. Pentru `while` şi `do-while` aceasta reprezintă o nouă evaluare a expresiei, iar pentru `for` înseamnă reiniţializarea ciclului, începând cu reevaluarea expresiei `e1` (vezi *supra*).

4.8. *goto*

Instrucţiunea `goto` efectuează un salt forţat la un anumit punct din program, denumit *etichetă* (*label*). Deoarece C este considerat un limbaj structurat, nu este recomandată folosirea acestei instrucţiuni, practic fiind posibilă întotdeauna evitarea ei. Ca exemplu, `goto` este folosită în cadrul nucleului Linux pentru a forţa compilatorul să genereze un cod mai eficient.

```
for(i = 0; i < 10; i++)
    if(x - i < 0)
goto ok;

ok:
...
```

Capitolul 5

POINTERI

Aceasta este închisă în memoria mea,
şi singur trebuie să găseşti cheia.
(Shakespeare, *Hamlet*, Act I, Scena V)

Prin definiţie, un *pointer* este o variabilă care conţine adresa unei alte variabile. Pointerii sunt foarte utilizaţi în C pentru că simplifică implementarea unor numeroase probleme şi conduc la generarea de cod obiect mai compact şi mai eficient.

Este posibilă adresarea variabilei conţinute de pointer în mod indirect. Presupunem că avem variabila `x` de tip `int`, iar `px` un pointer la tipul `int`. Operatorul unar „&" furnizează adresa la care se află o variabilă. Instrucţiunea:

```
px = &x;
```

atribuie variabilei `px` adresa la care este memorată variabila `x`. Vom spune că *`px` pointează pe `x`.*

Operatorul unar „*" returnează conţinutul variabilei memorate la adresa respectivă. În cazul nostru, instrucţiunea:

```
x = *px;
```

atribuie variabilei `x` valoarea memorată la adresa `px`. Declaraţia:

```
int *px;
```

declară că pointerul `px` memorează o valoare de tip `int`.

Iniţial, conţinutul variabilei de tip pointer `px` este vid. Pentru a putea folosi această variabilă, trebuie mai întâi alocat spaţiul de memorie necesar, cu ajutorul funcţiei de bibliotecă `malloc()`. Această funcţie primeşte ca argument dimensiunea spaţiului de memorie care se doreşte a fi alocat şi returnează adresa de unde începe zona de memorie alocată, în caz de succes, sau un pointer cu valoarea *NULL* (practic 0) atunci când alocarea nu a reuşit. Trebuie menţionat aici faptul că această funcţie returnează o variabilă de tip `void *`, adică un pointer de tip general, `void`. Pentru a se evita generarea de atenţionări (engl. *warnings*) de către

compilator, este necesară utilizarea operatorului *cast* pentru modificarea acestui tip în tipul pointerului `px` atunci când se doreşte ca `px` să primească drept valoare adresa zonei alocate cu `malloc()`:

```
px = (int *)malloc(2);
```

Operaţiunea descrisă mai sus se numeşte *alocare dinamică.*

Atunci când nu se cunoaşte mărimea tipului de variabilă (cum ar fi în cazul unei structuri, vezi capitolul 7), trebuie utilizată instrucţiunea `sizeof`:

```
double *d;

d = (double *)malloc(sizeof(double));
```

Eliberarea zonei de memorie ocupată de conţinutul unui pointer se face cu ajutorul funcţiei `free()`. Eliberarea acestor zone de memorie se face în mod automat la ieşirea din program.

```
free(d);
```

Pointerii rezolvă şi anumite inconveniente în folosirea parametrilor unei funcţii, deoarece aceştia sunt transmişi prin valoare, nefiind posibilă modificarea adreselor memorate prin intermediul lor de către funcţie. Transmiţând ca parametri variabile de tip pointer, adresele spre care pointează aceştia nu pot fi schimbate, dar conţinutul lor poate fi modificat fără nici o restricţie. Funcţia din exemplul următor interschimbă valorile parametrilor primiţi:

```
void swap(int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}
```

De asemenea, utilizarea pointerilor rezolvă problema că o funcţie nu poate returna decât o valoare, iar programatorul doreşte ca funcţia să poată returna mai multe. Această dilemă se poate rezolva trimiţând ca argumente funcţiei pointeri al căror conţinut va fi modificat de funcţie cu valorile care practic se doresc a fi returnate.

5.1. Tablouri şi pointeri

În C, există o relaţie foarte strânsă între pointeri şi tablouri, după cum vom observa în cele ce urmează.

Un tablou poate fi utilizat ca şi cum ar fi un pointer, iar un pointer poate fi indexat ca şi cum ar fi un tablou. Valoarea unei variabile de tip tablou reprezintă adresa primului element al tabloului:

```
int t[20];
int *pt;
int x;
```

În urma instrucţiunii:

```
pt = &t[0];
pt = t;
```

`pt` va pointa la primul element al tabloului `t`. Cele două atribuiri sunt echivalente. Instrucţiunea:

```
x = *pt;
```

atribuie lui `x` valoarea primului element al tabloului, iar:

```
x = *(pt + 1);
```

atribuie lui `x` valoarea celui de-al doilea element al tabloului. Dacă `pt` este un pointer, el poate fi accesat cu ajutorul indicilor sub forma `pt[i]`, ceea ce este echivalent cu `*(pt + i)`.

Pentru a ilustra mai bine aceste noţiuni de aşa-zisă *aritmetică a pointerilor*, vom rescrie funcţia `lungime_sir()` care calculează lungimea unui şir de caractere:

```
int lungime_sir(char *s)
{
    int n;

    for(n = 0; *s != '\0'; s++)
    n++;

    return n;
}
```

Se observă incrementarea lui `s`, operaţiune corectă cu un pointer, care nu afectează conţinutul acestuia, incrementând adresa `s`, aceasta memorând o copie

locală în cadrul funcţiei a parametrului actual de la apelul funcţiei. De remarcat că notaţia `char *s` este echivalentă cu `char s[]`. Dacă p este un pointer, atunci p++ desemnează următorul „obiect" pointat, indiferent de tipul acestuia.

Pentru a evidenţia uşurinţa programării cu pointeri, vom implementa o funcţie pentru copierea unui şir de caractere într-un alt şir, furnizând mai întâi varianta fără pointeri, iar apoi varianta cu pointeri:

```
void my_strcpy1(char s[], char t[])
{
    int i = 0;

    while((s[i] = t[i] != '\0')
       i++;
}

void my_strcpy2(char *s, char *t)
{
    while(*s++ = *t++)
    ;
}
```

5.2. Tablouri de pointeri

Pointerii pot fi utilizaţi şi în *tablouri de pointeri*. Spre exemplu, un tablou de pointeri poate constitui un text, fiecare linie a acestuia fiind reprezentată ca un pointer la primul său caracter. Vom prezenta în continuare un program de sortare lexicografică a acestor linii. Pointerii simplifică această operaţiune, deoarece atunci când două linii neordonate trebuie inversate, vor fi inversaţi doar pointerii corespunzători în tablou, nu şi conţinutul liniilor de text:

functii.h:

```
/*
 compara cele doua siruri primite ca argumente
 returneaza zero daca sunt identice, un numar pozitiv daca s1>s2 si
 un numar negativ daca s1<s2
*/
int strcmp(char *s1, char *s2);

/*
 returneaza lungimea sirului primit ca argument
*/
int strlen(char *s);
```



```
/*
 citeste de la tastatura linii de text, cel mult numarul specificat,
 pana cand intalneste EOF, memorandu-le in argumentul text
 returneaza numarul de linii introduse
*/
int readlines(char **text, int max);

/*
 sorteaza liniile textului primit ca argument
 returneaza zero in caz de succes, -1 in caz de eroare
*/
int sort(char **text, int n);
```

functii.c:

```
#include <stdio.h>

int strcmp(char *s1, char *s2)
{
    for(; *s1 == *s2; s1++, s2++)
       if(*s1 == '\0')
          return 0;

    return (*s1 - *s2)
}

int strlen(char *s)
{
    int n;

    for(n = 0; *s != '\0'; s++)
       n++;

    return n;
}

int readlines(char **text, int max)
{
    int len, nlines = 0;
    char s[255];

    while(nlines < max && (len = strlen(gets(s))) > 0)
    {
       if((text[nlines++] = (char *)malloc(len)) == NULL)
          /* daca alocarea de memorie a esuat */
          return -1;
       strcpy(text[nlines++], s);
    }

    return nlines;
}
```

```
int sort(char **text, int n)
{
    int i, j, dist;
    char *tmp;

    for(dist = n / 2; dist > 0; dist /= 2)
       for(i = dist; i < n; i++)
          for(j = i - dist; j >= 0; j -= dist)
          {
             if(strcmp(text[j], text[j + dist]) <= 0)
                break;
             tmp = text[j];
             text[j] = text[j + dist];
             text[j + dist] = tmp;
          }
}

void printlines(char **text, int n)
{
    int i = 1;

    while(--n >= 0)
       printf("%d: %s\n", i, *text++, i++);
}
```

main.c:

```
#include <stdio.h>
#define MAX_LINES 1000

int main()
{
    char *text[MAX_LINES];
    int i, nlines;

    nlines = readlines(text, MAX_LINES);
    sort(text, nlines);
    printlines(text, nlines);

    return 0;
}
```


5.3. Pointeri la funcţii

În C, există posibilitatea de a defini un pointer la o funcţie, care poate fi manipulat ca orice altă variabilă. Pentru exemplificare, vom modifica programul de sortare a liniilor unui text, prezentat mai sus, pentru a putea sorta orice fel de date, nu doar şiruri de caractere. Algoritmul de sortare va fi independent de operaţiile de comparare şi de inter-schimbare a liniilor de text, astfel încât, prin transmiterea acestor două funcţii ca argumente, se va putea realiza sortarea pe diferite criterii.

```
#include <stdio.h>
#define MAX_LINES 1000

int nrcmp(char *s1, char *s2)
{
    double n1, n2;

    n1 = atof(s1);
    n2 = atof(s2);
    if(n1 < n2)
       return -1;
    else
       if(n1 > n2)
          return 1;

    return 0;
}

void schimba(char **s1, char **s2)
{
    char *tmp;

    tmp = *s1;
    *s1 = *s2;
    *s2 = tmp;
}

int sort(char **text, int n, int (*compara)(), (*schimba)())
{
    int i, j, dist;

    for(dist = n / 2; dist > 0; dist /= 2)
       for(i = dist; i < n; i++)
          for(j = i - dist; j >= 0; j -= dist)
          {
             if((*compara)(text[j], text[j + dist]) <= 0)
                break;
          (*schimba)(&text[j], &text[j + dist]);
          }
}
```

```
int main(int argc, char **argv)
{
    char *text[MAX_LINES];
    int i, nlines, numeric = 0;   /* 1 daca valorile sunt numerice */

    if(argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n')
      numeric = 1;

    nlines = readlines(text, MAX_LINES);
    if(numeric)
      sort(text, nlines, nrcmp, schimba);
    else
      sort(text, nlines, strcmp, schimba);
    printlines(text, nlines);

    return 0;
}
```

Declaraţia `int (*compara)()` desemnează că argumentul `compara` este un pointer la o funcţie care returnează un `int`. Declaraţia `(*schimba)()` indică un pointer la o funcţie care returnează un `void`.

Capitolul 6

STRUCTURAREA PROGRAMELOR: FUNCŢIILE

Natura utilizează doar cele mai lungi căi pentru a-şi ţese şablonul, astfel că fiecare mică bucată de ţesătură dezvăluie organizarea întregii tapiserii.
(Richard Feynman)

6.1. Recursivitatea

În limbajul C, funcţiile pot fi apelate recursiv. Aceasta înseamnă că o funcţie se poate apela pe ea însăşi. Atunci când o funcţie se autoapelează, fiecare apel generează alte instanţe ale variabilelor declarate în interiorul funcţiei. Recursivitatea este utilă în special pentru prelucrarea structurilor de date recursive, cum ar fi arborii sau grafurile.

Trebuie avut în vedere faptul că utilizarea de funcţii recursive duce la consum ridicat de memorie, deoarece pentru fiecare apel de funcţie se alocă spaţiu pentru stiva funcţiei precum şi pentru variabilele declarate în interiorul ei.

Exemplul următor implementează funcţia `calcul`, care decrementează un număr întreg până când acesta devine zero:

```
#include <stdio.h>

void calcul(int *contor)
{
    if(*contor == 0)
      return;
    else
      {
      (*contor)--;
      calcul(contor);
      }
}

int main()
{
    int i;

    printf("Introduceti numarul de cicluri: ");
    scanf("%d", &i);
```

```
    calcul(&i);
    printf("Valoarea lui i este: %d\n", i);

    return 0;
}
```

6.2. Funcţia *main()*

S-a putut observa în multe exemple de până acum apelul `main()`. Aceasta nu este altceva decât o funcţie specială, care defineşte programul principal. Apelul acestei funcţii începe o dată cu execuţia programului şi încetează la apelul instrucţiunii `return` din cadrul funcţiei `main()` sau la încheierea programului.

În mod normal, funcţia `main()` trebuie să returneze un număr întreg, reprezentând rezultatul execuţiei programului. În general, aceasta este zero pentru o execuţie normală.

Funcţia `main()` poate primi argumente de forma:

```
int main(int argc, char **argv)
```

unde `argc` este numărul de parametri primiţi de program la apelarea acestuia, iar `argv` un tablou de şiruri de caractere (vezi capitolul 5) conţinând parametri. Primul element al tabloului, `argv[0]`, reprezintă numele executabilului asociat programului, inclusiv calea completă în care acesta se află. Programul de mai jos va afişa toţi parametrii primiţi în linia de comandă.

```
main(int argc, char **argv)
{
    int i;

    for(i = 1; i < argc; i++)
      printf("Parametrul %d: %s\n", i, argv[i]);
}
```

Presupunând că fişierul executabil corespunzător acestui program este denumit `afisparam`, apelul:

```
afisparam param1 param2
```

va afişa la ieşirea standard:

```
Parametrul 1: param1
Parametrul 2: param2
```

Capitolul 7

STRUCTURI

Regula este: îl faci cât de simplu poţi,
dar nu mai simplu de-atât.
(Albert Einstein)

O *structură* este o colecţie de date de tipuri diferite. O structură se declară utilizând sintaxa:

```
struct nume_structura {
    declaratii membri
    ...
}
```

De exemplu:

```
struct student {
    char *nume;
    char *prenume;
    char *adresa;
    char *telefon;
}
```

Pentru a declara o variabilă de tip `student` şi a o iniţializa:

```
struct student s = { "Buraga", "Sabin", "Iasi, str. Pacurari",
    "0744123456" };
```

Membrii structurii pot fi accesaţi prin construcţii de forma:

```
nume_structura.nume_membru
```

Spre exemplu:

```
printf("Nume si prenume: %s %s\n", s.nume, s.prenume");
```

În mod evident, pot fi declaraţi şi pointeri la structuri. Pentru accesarea membrilor trebuie folosit în acest caz operatorul `->`:

```
struct student *s;

s = (struct student *)malloc(sizeof(struct student));
s->nume = (char *)malloc(10);
strcpy(s->nume, "Buraga");
```

O structură poate conţine şi membri de tip structură:

```
struct data {
    int zi, luna, an;
};

struct student {
    char *nume;
    char *prenume;
    char *adresa;
    char *telefon;
    struct data data_nastere;
} s;
```

Ultima declaraţie, cea a structurii `student`, conţine şi o definiţie, cea a variabilei `s` de tip `student`. Membrii structurii `data` se accesează astfel:

```
s.data_nastere.zi
```

Manipularea structurilor este supusă unui număr de restricţii. Astfel, structurile nu pot fi asignate sau copiate ca un întreg, elementele acestora trebuind copiate individual de către programator. Este permisă utilizarea structurilor în cadrul argumentelor funcţiilor, precum şi ca tip de dată returnată de o funcţie:

```
struct student citire_student();

int verificare_student(struct student *s);
```

Conţinutul unei variabile de tip structură poate fi iniţializată astfel:

```
struct student s = { "Buraga", "Sabin", "Iasi", "0744123456",
    { 07, 01, 1974 } };
```


7.1. Structuri cu autoreferire

Autoreferirea desemnează posibilitatea ca o structură să aibă ca membru un pointer la o structură de acelaşi tip cu ea:

```
struct nod {
    int val;
    struct nod *stanga, *dreapta;
};

#include <stdio.h>
#include <string.h>

struct nod {
    char *linie;
    struct nod *stanga, *dreapta;
};

struct nod *inserare_linie(struct *rad, char *sir)
{
    struct nod *nou;

    if(rad == NULL)
       {
       /* alocam memorie pentru un nod */
       if((nou = (struct nod *)malloc(sizeof(struct nod))) == NULL)
          /* daca alocarea de memorie a esuat */
          return -1;

       /* alocam memorie pentru sir */
       if((nou->linie = (char *)malloc(strlen(sir))) == NULL)
          /* daca alocarea de memorie a esuat */
          return -1;

       strcpy(rad->linie, sir);
       rad->stanga = rad->dreapta = NULL;
       }
    else
       {
       if(strcmp(sir, rad->linie) < 0)
          rad->stanga = inserare_linie(rad->stanga, sir);
       else
          rad->dreapta = inserare_linie(rad->dreapta, sir);
       }

    return rad;
}

void afiseaza_linie(struct nod *rad)
{
    if(rad->stanga)
```

```
        afiseaza_linie(rad->stanga);
    printf("%s\n", rad->linie);
    if(rad->dreapta)
        afiseaza_linie(rad->dreapta);
}

int main()
{
    int len;
    char s[256];
    struct nod *rad = NULL;

    while((len = strlen(fgets(s, 255, stdio))) > 0)
        rad = inserare_linie(rad, s);
    afiseaza_linie(rad);

    return 0;
}
```

7.2. *typedef*

Într-un mod similar cu macrodefinițiile, limbajul C permite crearea unor nume noi pentru diferite tipuri de date, denumite `typedef`. Astfel, declarația:

```
typedef char* SIR;
```

face ca numele de tip `SIR` să fie sinonim pentru `char *`. Vom utiliza `typedef` pentru structura arborelui de mai sus:

```
typedef struct nod {
    char *linie;
    struct nod *stanga, *dreapta;
} NOD_ARBORE;
```

Declarațiile `typedef` sunt utile în primul rând pentru tipuri de date care pot fi dependente de platformă, modificările trebuind efectuate doar asupra declarației `typedef`, şi în al doilea rând pentru a creşte claritatea unui program atunci când în cadrul acestuia intervin structuri mai complicate de date.

Capitolul 8

PREPROCESORUL C

Un algoritm trebuie să fie văzut
pentru a putea fi crezut.
(Donald Knuth)

C permite utilizarea unor extensii de limbaj prin intermediul *macrourilor*. Aceste macrouri sunt analizate înainte de compilare, în faza de *preprocesare*.

Includerea fişierelor-antet

Pentru includerea fişierelor-antet (dar nu exclusiv) se pot utiliza următoarele două macrouri:

```
#include "fisier"
#include <fisier>
```

Practic, un astfel de macrou va fi înlocuit în faza de preprocesare cu conținutul fişierului specificat. Prima variantă a macroului face presupunerea că fişierul specificat se găseşte în directorul curent, iar cea de-a doua variantă caută fişierul în căile cunoscute de compilator (în cazul `GCC`, folosind opțiunea `-I`).

Macrosubstituțiile

Macrosubstituirea desemnează înlocuirea unui nume cu un şir de caractere. Aceasta se poate realiza prin definiții de forma:

```
#define MAX 100
```

Atunci când în codul-sursă va fi întâlnit cuvântul `MAX`, acesta va fi înlocuit cu `100`. Se pot defini de asemenea şi macrouri cu argumente, ca în exemplul de mai jos, care calculează maximul dintre două numere:

```
#define max(x, y) (x > y ? y : x)
```

Compilare condiţionată

O expresie de forma:

```
#if constanta
```

verifică dacă respectiva constantă este non-zero. O expresie de forma:

```
#ifdef simbol
```

verifică dacă simbolul este definit (printr-o directivă `#define`). În mod similar, o expresie de genul:

```
#ifndef simbol
```

verifică dacă simbolul nu este definit. Toate aceste trei directive preprocesor pot fi urmate de linii de cod-sursă care pot fi urmate opţional de `#else` şi apoi obligatoriu de `#endif`. În cazul în care condiţia este adevărată, liniile de cod dintre `#else` şi `#endif` sunt ignorate. În cazul în care condiţia este falsă, liniile dintre directivă şi `#else` sunt ignorate.

Echivalentă cu expresia `#ifdef simbol` este construcţia:

```
#if defined(simbol)
```

Această formă este mai permisivă, în sensul că sunt valide exprimări de genul:

```
#if defined(simbol1) || defined(simbol2)
```

Capitolul 9

BIBLIOTECILE STANDARD C

scanf e tare.
(Al. Viro pe lista `linux-kernel`)

După cum am mai spus, operaţiunile de intrare/ieşire sunt implementate în biblioteca standard C şi sunt accesibile prin intermediul funcţiilor din cadrul acesteia. Pentru a putea folosi această bibliotecă, trebuie inclus fişierul-antet `stdio.h` printr-o directivă `#include <stdio.h>`. O serie de funcţii au fost utilizate în programele prezentate în cadrul acestei părţi a lucrării de faţă.

Ieşirea formatată: *printf*

Sintaxa funcţiei `printf` este următoarea:

```
printf(const char *format, arg1, arg2, ...)
```

`printf` converteşte, formatează şi tipăreşte la ieşirea standard valorile argumentelor *arg1*, *arg2*, ..., conform formatului specificat prin primul argument. Acest format conţine caractere obişnuite, care vor fi copiate la ieşire, şi caractere de control, care determină modul de conversie şi tipărirea următoarelor argumente. Fiecare specificaţie de conversie este introdusă prin caracterul „`%`" şi încheiată printr-un caracter de conversie.

Între „`%`" şi caracterul de conversie pot fi:

- un semn minus „`-`", care semnifică alinierea la stânga a argumentului convertit;
- un număr care specifică lungimea minimă a şirului. Argumentul va fi aliniat la dreapta şi va fi completat cu spaţii albe până la lungimea câmpului. Dacă dimensiunea câmpului a fost prefixată cu un „`0`", caracterul de completare va fi „`0`".

Caracterele de conversie sunt următoarele:

- `d` argumentul este convertit în zecimal;
- `o` argumentul este convertit în octal fără semn;
- `x` argumentul este convertit în hexazecimal fără semn;
- `u` argumentul este convertit în zecimal fără semn;

- `c` argumentul este considerat ca fiind un singur caracter;
- `s` argumentul este un şir de caractere. Caracterele din cadrul argumentului corespunzător acestui format de conversie sunt preluate până la întâlnirea caracterului `\0` sau până când este atinsă lungimea specificată;
- `f` argumentul este considerat în virgulă mobilă, implicit cu 6 cifre după virgulă.

Asemănătoare cu funcţia `printf` sunt funcţiile:

- `sprintf(char *sir, const char *format, arg1, arg2, ...)` trimite ieşirea către şirul de caractere specificat;
- `fprintf(FILE *fisier, const char *format, arg1, arg2, ...)` trimite ieşirea către identificatorul de fişier specificat (vezi *infra*).

Intrarea formatată: *scanf*

`scanf` citeşte caractere de la intrarea standard, le interpretează conform formatului specificat şi le memorează în argumentele următoare, care practic sunt pointeri care indică adresele unde vor fi memorate datele convertite. Sintaxa funcţiei `scanf` este următoarea:

```
int scanf(const char *format, arg1, arg2, ...)
```

Formatul primului argument este identic cu cel descris la funcţia `printf`. Funcţia `scanf` returnează numărul de „obiecte" citite de la intrare. Important de remarcat este faptul că spaţiul de memorie alocat pentru pointerii trimişi ca argumente trebuie să fie destul de mare pentru a putea memora datele citite.

```
#include <stdio.h>

int main()
{
    int x, y;

    printf("Introduceti cele doua valori: ");
    scanf("%d %d", &x, &y);
    printf("%d + %d = %d\n", x, y, x + y);

    return 0;
}
```

Asemănătoare cu funcţia `scanf` sunt funcţiile:

- `sscanf(char *sir, const char *format, arg1, arg2, ...)` citeşte date din şirul de caractere specificat;
- `fscanf(FILE *fisier, const char *format, arg1, arg2, ...)` citeşte date din identificatorul de fişier specificat (vezi *infra*).

Funcţii pentru citirea şi scrierea de caractere

- `int getchar()` citeşte un caracter de la intrarea standard;
- `int fgetc(FILE *f)` citeşte un caracter din fişierul specificat;
- `int putchar(int c)` scrie un caracter la ieşirea standard;
- `int fputc(int c, FILE *f)` scrie un caracter în fişierul specificat.

Funcţii pentru citirea şi scrierea de şiruri de caractere

- `char *gets(char *s)` citeşte un şir de caractere de la intrarea standard, memorându-l în argumentul `s`. Nu face vreo verificare asupra dimensiunii spaţiului alocat şirului `s`;
- `char *fgets(char *s, int max, FILE *f)` citeşte din fişierul `f` un şir de caractere a cărui lungime nu va depăşi numărul maxim de caractere, specificat ca argument;
- `int puts(char *s)` scrie un şir de caractere la ieşirea standard;
- `int fputs(char *s, FILE *f)` scrie un şir de caractere în fişierul `f`.

Lucrul cu fişiere

Înainte de a putea citi sau scrie într-un fişier, acesta trebuie deschis cu ajutorul funcţiei `fopen`. Această funcţie returnează un pointer la o structură de tip `FILE`, care conţine informaţii despre fişierul deschis, utilizate intern de către bibliotecă:

```
FILE *fopen(const char *nume_fisier, const char *mod)
```

Modul de deschidere a fişierului se alege în funcţie de modul în care se intenţionează a fi utilizat:

- `r` doar citire;
- `w` scriére;
- `a` adăugare.

Dacă fişierul nu există şi este deschis pentru scriere sau adăugare, acesta va fi creat. Dacă se deschide un fişier existent pentru scriere, conţinutul vechi va fi şters. Dacă la deschidere apare o eroare, funcţia `fopen` va returna valoarea `NULL`.

Închiderea unui fişier atunci când nu mai este folositor se realizează cu ajutorul funcţiei `fclose`, care primeşte ca argument pointerul la `FILE` care specifică fişierul respectiv. Conţinutul acestuia va fi eliberat.

În momentul începerii execuţiei unui program C, sunt deschise în mod automat trei fişiere: intrarea standard (`stdin`), ieşirea standard (`stdout`) şi ieşirea

standard pentru erori (`stderr`). Acestea pot fi folosite ca orice alt fişier, dar nu pot fi deschise sau închise (ele sunt practic constante).

Vom realiza în cele ce urmează un program care simulează comanda `cat`, pentru concatenarea a două fişiere:

```
#include <errno.h>
#include <stdio.h>

void copie(FILE *f)
{
    int c;

    while((c = fgetc(f)) != EOF)
      putc(c, stdout);
}

int main(int argc, char **argv)
{
    FILE *f;

    if(argc == 1)/* nu am primit argumente, utilizam intrarea
                    standard */
      copie(stdin);
    else
      /* va fi afisat intai continutul fisierului precizat prin
         argv[1], apoi continutul fisierului argv[2]
      */
      while(--argc > 0)
        if((f = fopen(*++argv, "r")) == NULL)
          {
          fprintf(stderr, "cat: nu pot deschide fisierul `%s'\n",
              *argv);
          return -ENOENT;      /* fisierul nu exista */
          }
        else
          {
          copie(f);
          fclose(f);
          }

    return 0;
}
```

Vom prezenta mai jos un program care extrage fişierele conţinute într-o arhivă `tar`:

```
/*
 untar v1.0

 Dezarhiveaza o arhiva tar furnizata ca parametru
*/
```



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
    FILE *in, *out;
    char buff[256];
    char *currdir, *name;
    char c;
    unsigned long i, currpos = 0;

    printf("untar v1.0 -- Dezarhiveaza o arhiva tar\n\n");
    if(argc != 2)
      {
      printf("Numar invalid de parametri.\n");
      printf("Mod de utilizare: untar fisier\n");
      return -1;
      }
    if((in = fopen(argv[1], "r")) == NULL)
      {
      printf("untar: nu pot deschide fisierul `%s'\n");
      return -1;
      }
    while(!feof(in))
      {
      fseek(in, currpos, SEEK_SET);
      i = 0;
      /* determina numele fisierului sau directorului continut in
         arhiva
      */
      while((c = fgetc(in)) != '\0')
        buff[i++] = c;
      if(i == 0)
        break;
      if(buff[i - 1] == '/')
        {
        buff[i - 1] = '\0';
        currdir = buff;
        /* daca este un director, il creez */
        if(mkdir(currdir, 0755))
          printf("untar: nu pot crea directorul `%s'\n", currdir);
        else
          printf("%s/\n", currdir);
        strcat(currdir, "/");
        }
      else
        {
        buff[i] = '\0';
        name = buff;
        printf("%s\n", name);
        /* daca este un fisier, il scriu pe disc */
```

```
        if((out = fopen(name, "w")) == NULL)
           {
           printf("untar: nu pot deschide fisierul `%s' pentru
              citire\n", name);
           fclose(in);
           return -1;
           }
        currpos += 0x200;
        fseek(in, currpos, SEEK_SET);
        while((c = fgetc(in)) != '\0')
           {
           if(c == EOF)
              {
              printf("untar: sfarsit neasteptat de fisier\n");
              fclose(in);
              fclose(out);
              return -1;
              }
           fputc(c, out);
           }
        fclose(out);
        currpos = ftell(in);
        while((currpos % 0x200) != 0)
           currpos++;
        currpos -= 0x200;
        }
     currpos += 0x200;
     }
     fclose(in);

     return 0;
}
```

Întrucât scopul lucrării de faţă nu este învăţarea limbajului C, recomandăm cititorului să parcurgă anexele 1 şi 2, care conţin lista tuturor funcţiilor de bibliotecă prevăzute de standardele ANSI şi POSIX, precum şi resursele bibliografice.

Capitolul 10

PROBLEME PROPUSE

1. Să se implementeze funcţia `char *elimina_caracter(const char *s, const char c)`, care elimină toate apariţiile caracterului `c` din şirul `s`.
2. Să se implementeze funcţia `double atof(const char *s)`, care converteşte un şir de caractere într-un număr de tip `double` (vezi implementarea funcţiei `atoi()` în paragraful 4.5).
3. Să se modifice programul prezentat în subcapitolul 5.2 pentru sortarea unor linii de text, astfel încât să sorteze liniile de text primite de la intrarea standard.
4. Să se scrie un program care să definească structura pentru un arbore şi să implementeze operaţiile uzuale pentru aceasta (creare, căutare, ştergere, inserare, parcurgere). Nodurile arborelui vor conţine elemente de tip `int`.
5. Să se modifice programul precedent pentru lucrul cu arbori, pentru a putea salva, respectiv încărca informaţiile într-un fişier. Nodurile arborelui vor conţine elemente de tip şir de caractere.
6. Să se scrie o variantă simplificată a utilitarului `bc`, adică un program care să aştepte de la tastatură expresii de forma *operand_stanga operator operand_dreapta* şi să afişeze rezultatul operaţiei, acceptând operatorii „+”, „-”, „/” şi „%”.
7. Să se implementeze o variantă a comenzii `grep` care caută într-un fişier pe baza unor criterii un subşir de caractere, liniile găsite fiind afişate. Opţiunile suportate vor fi:
   –v afişează doar liniile care nu satisfac criteriul;
   –c afişează numai numărul liniilor găsite, nu şi liniile;
   –i ignoră literele mici de cele mari.

   Subşirul de căutat va putea include următoarele meta-caractere:
   - „*” înlocuieşte mai multe caractere;
   - „?” înlocuieşte un singur caracter.
8. Utilizând structura definită în capitolul 7, să se realizeze un program care să permită evidenţa studenţilor dintr-o facultate. Programul trebuie să implementeze şi salvarea, respectiv încărcarea datelor de pe disc.
9. Să se scrie un program denumit `aat`, care să aibă sintaxa `aat [` *timp comanda* `| delete ]`. Acesta va executa comanda la timpul specificat (poate fi de forma *AAAALLZZOOMM*). Dacă este menţionată opţiunea `delete`, va fi ştearsă planificarea comenzii la momentul specificat. Fără parametri, vor fi afişate comenzile planificate a fi executate.

10. Să se implementeze un program care simulează comanda `tar`, care să gestioneze un fişier-arhivă care include mai multe fişiere. Se vor implementa opţiunile:
`-c` creează o arhivă care va îngloba conţinutul fişierelor din directorul curent;
`-x` dezarhivează arhiva, restaurând fişierele conţinute de aceasta;
`-t` listează conţinutul arhivei.

Pentru simplificarea programului, nu se va căuta recursiv în subdirectoare. De asemenea, în arhivă vor fi memorate doar numele şi dimensiunea fişierelor.

Partea a doua

PROGRAMAREA ÎN LIMBAJUL C++

Capitolul 1

SCURT ISTORIC

Începutul este momentul în care trebuie acordată
cea mai mare atenție corectitudinii echilibrelor.
(Frank Herbert, *Dune*)

Așa cum am văzut în partea I a acestei cărți, C este un limbaj procedural, structurat. Creșterea complexității programelor a făcut însă necesară elaborarea altor tipuri de limbaje. Astfel, destinate inteligenței artificiale, au apărut limbajele care au la bază noțiunea de „*cadru*" și cele care pleacă de la ideea de „*actor*". Primele implementează *operații* asupra unor *modele de entități*; celelalte presupun faptul că *obiectele* nu sunt simple elemente pasive asupra cărora se fac anumite prelucrări, ci, dimpotrivă, că menirea acestor *obiecte* constă în a realiza prelucrările asupra lor înșile.

De aici a pornit ideea de a grupa structurile de date cu operațiile care prelucrează respectivele date. Astfel s-au născut noțiunile de *obiect* și de *clasă*. Proiectarea de programe utilizând clase se numește *programare orientată-obiect* (*OOP, Object Oriented Programming*).

Primele limbaje orientate-obiect au fost *SIMULA* (1965) și *SIMULA-2* (1967). În anii '70 a apărut și celebrul limbaj *SMALLTALK*. Cel mai mare dezavantaj al acestora a fost faptul că au fost proiectate ca limbaje de sine stătătoare, având o răspândire relativ redusă. Din acest motiv, puțini programatori erau dispuși în acea vreme să renunțe la limbajele consacrate doar pentru a lucra obiectual.

În anul 1980, Bjarne Stroustrup a conceput limbajul „*C with Classes*" („*C cu clase*"). Acest limbaj a dus la îmbunătățirea C-ului prin adăugarea unor noi facilități, printre care și lucrul cu clase. În vara lui 1983, C-with-classes a pătruns și în lumea academică și a instituțiilor de cercetare. Astfel, acest limbaj a putut să evolueze datorită experienței acumulate de către utilizatorii săi. Denumirea finală a acestui limbaj a fost *C++*.

Succesul extraordinar de care se bucură limbajul C++ a fost asigurat de faptul că a extins cel mai popular limbaj al momentului, C. Programele scrise în C funcționează și în C++, și ele pot fi transformate în C++ cu eforturi minime.

Cele mai recente etape în evoluția acestui limbaj sunt limbajele *JAVA*, realizat de firma *SUN*, și *C#*, propus de *Microsoft*, care nu sunt altceva decât limbaje C++ puțin modificate și extinse.

Interesantă este următoarea afirmaţie a lui Stroupstrup: „*utilizatorii au început să folosească C++ înainte ca specialiştii să aibă timpul necesar să-i instruiască pentru a-l folosi cu randament maxim*". Într-adevăr, s-a constatat că o mare parte dintre compilatoarele de C++ existente nu sunt folosite decât pentru dezvoltarea de software structurat, şi nu orientat-obiect (altfel spus, se lucrează în C pe un compilator de C++).

Programatorii au descoperit ulterior că aplicaţiile orientate-obiect sunt mai uşor şi mai rapid de scris, şi nu în ultimul rând mai uşor de înţeles.

Capitolul 2

NOŢIUNILE DE BAZĂ ALE PROGRAMĂRII ORIENTATE-OBIECT

Procesul programării are mai multe faze:
se începe cu definirea conceptelor,
după care se trece la stabilirea relaţiilor dintre ele.
Abia după aceea se poate trece la scrierea codului.
(Bjarne Stroustrup)

2.1. Premisele limbajelor orientate-obiect

În ultimii ani, programarea orientată-obiect a devenit foarte populară, mai ales datorită avantajelor sale utile dezvoltării proiectelor actuale, ce devin din ce în ce mai complexe. Acest stil de programare propune împărţirea aplicaţiilor în mai multe module, astfel încât cel ce dezvoltă un modul nu trebuie să cunoască detaliile de implementare a altor module.

Trebuie totodată să amintim că programarea orientată-obiect este un concept foarte natural. În lumea înconjurătoare, zi de zi, în orice moment, avem de-a face cu *Obiecte*. În jurul nostru se găsesc o multitudine de obiecte, *interconectate, comunicând unele cu altele* într-un fel sau altul.

Domeniul în care acest stil de programare s-a dovedit cel mai util este dezvoltarea interfeţelor utilizator şi a aplicaţiilor bazate pe acestea.

Programarea structurată este bazată pe ecuaţia enunţată de Niklaus Wirth:

Structuri de date + Algoritmi = Program

Programarea structurată a fost o etapă ce a trebuit depăşită, deoarece este deficitară în ceea ce priveşte posibilitatea reutilizării programelor, scalabilităţii şi extinderii unor module de program, atribute de neînlocuit în realizarea aplicaţiilor complexe. Principala deficienţă a programării structurate constă în tratarea separată a algoritmilor şi a structurilor de date ce se prelucrează. De obicei, în natură, o entitate este caracterizată atât prin structură, cât şi printr-un anumit comportament. În mod normal, obiectele evoluează în timp, adeseori modificându-şi structura şi funcţionalitatea.

2.2. Concepte fundamentale

Ideea de bază de la care pleacă programarea orientată-obiect este de a grupa structurile de date cu operaţiile care prelucrează respectivele date. Un asemenea ansamblu poartă denumirea de *clasă*. Proiectarea de programe utilizând clase se numeşte *programare orientată-obiect* (*OOP*).

În mod frecvent, pentru structurile de date se utilizează denumirea de *date membre* sau *câmpuri*, iar pentru procedurile ce prelucrează aceste date, termenul de *funcţii membre* sau *metode*.

În analogie cu ecuaţia programării structurate, se poate considera ca valabilă următoarea relaţie:

Date + Metode = Clasă

Această relaţie este bazată pe principiul fundamental al *încapsulării datelor*, conform căruia accesul la datele membre se poate face numai prin intermediul setului de metode asociat. Acest principiu determină o abstractizare a datelor în sensul că o clasă este caracterizată complet de specificaţiile metodelor sale, detaliile de implementare fiind ascunse utilizatorului. Acest aspect este hotărâtor în cazul proiectelor complexe, de dimensiuni mari, care nu pot fi realizate decât cu ajutorul unor echipe de programatori. Aplicaţiile pot fi împărţite cu uşurinţă în module, astfel încât cel ce dezvoltă un modul nu trebuie să cunoască detaliile de implementare a celorlalte module. Consecinţele imediate sunt scăderea timpului de dezvoltare a aplicaţiilor, simplificarea activităţii de întreţinere a modulelor şi creşterea calităţii programelor.

Privind limbajele orientate-obiect ca pe o evoluţie a limbajelor structurate, constatăm că noţiunea de clasă este o generalizare a noţiunii de structură de date. O clasă descrie un ansamblu de obiecte similare. Un obiect este aşadar o variabilă de un anumit tip. În mod uzual, se foloseşte exprimarea că *un obiect este instanţierea unei clase*.

Un alt concept important în cadrul programării orientate-obiect este cel de *polimorfism*, care se referă la posibilitatea de a opera cu mai multe variante ale unei funcţii, care efectuează o anumită operaţie specifică pentru anumite obiecte.

Evoluţia şi ierarhizarea claselor de obiecte se bazează pe conceptul de *moştenire*. Astfel, procedeul numit *derivare* permite definirea unei noi clase (*clasă derivată*) pornind de la o clasă existentă (*clasă de bază*), prin adăugarea de noi date şi metode membre, eventual prin redefinirea unor metode ale clasei de bază. Clasa derivată moşteneşte de la clasa de bază structura de date şi metodele aferente. Este posibilă totodată şi derivarea unei clase din mai multe clase de bază, această operaţie fiind denumită *moştenire multiplă*.

Aşadar, dintr-o clasă de bază pot fi derivate mai multe clase şi fiecare clasă derivată poate deveni la rândul ei o clasă de bază pentru alte clase derivate. Se poate astfel realiza o *ierarhie de clase*, care să modeleze sisteme complexe. Construirea ierarhiei de clase constituie activitatea fundamentală de realizare a unei aplicaţii orientate-obiect, reprezentând în fapt faza de proiectare a respectivului sistem.

În capitolele următoare vom discuta despre implementarea acestor concepte în limbajul C++.

Capitolul 3

CLASE

Limbajul C îți permite să te împuști
singur cu ușurință în picior;
C++ face acest lucru mai dificil,
însă atunci când se întâmplă
îți spulberă întreg piciorul.
(Bjarne Stroustrup)

3.1. Declararea claselor

O sintaxă simplificată a declarării unei clase este următoarea:

```
class NumeClasa
{
    ...
    declaratii variabile membre
    ...
    declaratii functii membre
    ...
}
```

După cum se poate observa din această alcătuire a declarației, clasa este asemănătoare cu o structură din limbajul C, dar care poate avea în componența sa membri atât de tip variabilă, cât și de tip funcție. Așa cum spuneam și în capitolul precedent, pentru datele din interiorul clasei se utilizează de obicei termenul de *date membre*, iar pentru funcții, denumirea de *funcții membre* sau *metode*. Spunem așadar că o clasă permite *încapsularea* în interiorul său a datelor și a funcțiilor membre.

Întocmai ca în limbajul C, pentru a putea utiliza efectiv un tip de date (în cazul de față o clasă), trebuie să definim o variabilă de acel tip. Într-un mod similar declarației:

```
int i;
```

putem scrie:

```
NumeClasa variabila
```


Vom considera de acum încolo că *variabila* poartă numele de *obiect*. Exprimarea uzuală este că *un obiect este o instanță a unei clase*.

3.2. Membrii unei clase

Accesarea membrilor unei clase se face ca în cazul structurilor din limbajul C:

```
obiect.VariabilaMembra = valoare
```

pentru *accesul* la o variabila membră, și:

```
obiect.FunctieMembra()
```

pentru *apelarea* unei funcții membre.

Pentru exemplificare să considerăm o implementare a noțiunii de punct. Ca variabile membre avem nevoie doar de coordonatele `x` și `y` care definesc poziția în spațiu a unui punct. Declarăm o funcție care calculează aria dreptunghiului având colțurile `(0, 0)` și `(x, y)`.

```
class Point
{
    unsigned x, y;
    unsigned long Arie()
    {
       return x * y;
    };
    unsigned GetX();
    unsigned GetY();
    void SetX(unsigned X);
    void SetY(unsigned Y);
};

unsigned Point::GetX()
{
    return x;
}

unsigned Point::GetY()
{
    return y;
}

void Point::SetX(unsigned X)
{
    x = X;
}
```

```
void Point::SetY(unsigned Y)
{
    y = Y;
}
```

Am folosit un operator nou, specific C++, `::`, numit *operator de rezoluţie*, numit şi *operator de acces* sau de *domeniu*. El permite accesul la un identificator, dintr-un bloc în care acesta nu este vizibil datorită unei alte declaraţii locale. Un exemplu de folosire este următorul:

```
char *sir = "variabila globala";
void functie()
{
    char *sir = "variabila locala";
    printf("%s\n", ::sir); // afiseaza variabila globala
    printf("%s\n", sir); // afiseaza variabila locala
}
```

Pentru definiţiile funcţiilor membre aflate în afara declaraţiei clasei este necesară specificarea numelui clasei urmat de acest operator înaintea numelui fiecărei funcţii, indicând faptul că funcţia are acelaşi domeniu ca şi declaraţia clasei respective şi este membră a ei, deşi este definită în afara declaraţiei.

O întrebare care poate apărea se leagă de motivul pentru care am realizat funcţiile `GetX()`, `GetY()`, `SetX()`, `SetY()`, când puteam utiliza direct variabilele membru `x` şi `y`. Răspunsul este furnizat de una din regulile programării C++, adoptată în general de către specialişti, care cere protecţia variabilelor membru (a se vedea capitolul următor), acestea neputând fi accesate decât prin intermediul unor funcţii, care au rolul de *metode de prelucrare a datelor încapsulate în interiorul clasei*. Această regulă reprezintă principiul fundamental al *încapsulării datelor*.

3.3. Crearea şi distrugerea obiectelor

Să considerăm următorul program C++:

```
void main()
{
    Point p;
}
```

În momentul definirii variabilei `p`, va fi alocat automat spaţiul de memorie necesar, acesta fiind eliberat la terminarea programului. În exemplul de mai sus,

variabila *p* este de tip static. În continuare vom modifica acest program pentru a folosi o variabilă dinamică (pointer).

```
void main()
{
    Point *p;

    p = new Point;
    p->x = 5;
    p->y = 10;
    printf("Aria = %d\n", p->Aria());
    delete p;
}
```

Se poate observa utilizarea unor operatori pe care nu îi cunoaştem din limbajul C: `new` şi `delete`. C++ introduce o metodă nouă pentru gestiunea dinamică a memoriei, similară celei utilizate în C (`malloc()` şi `free()`), dar superioară şi special construită pentru programarea orientată-obiect. Operatorul *new* este folosit pentru alocarea memoriei, iar sintaxa acestuia este:

```
variabila = new tip;
variabila = new tip(valoare_initiala);
variabila = new tip[n];
```

Este uşor de intuit că prima variantă alocă spaţiu pentru *variabilă* dar nu o iniţializează, a doua variantă îi alocă spaţiu şi o iniţializează cu valoarea specificată, iar a treia alocă un tablou de dimensiune *n*. Acest operator furnizează ca rezultat un pointer conţinând adresa zonei de memorie alocate, în caz de succes, sau un pointer cu valoarea *NULL* (practic 0), atunci când alocarea nu a reuşit.

Eliminarea unei variabile dinamice şi eliberarea zonei de memorie aferente se realizează cu ajutorul operatorului *delete*. Sintaxa acestuia este:

```
delete variabila;
```

Deşi aceşti doi operatori oferă metode flexibile de gestionare a obiectelor, există situaţii în care aceasta nu rezolvă toate problemele (de exemplu, obiectele care necesită alocarea unor variabile dinamice în momentul creării lor). De aceea, pentru crearea şi distrugerea obiectelor în C++ se folosesc nişte funcţii membre speciale, numite *constructori* şi *destructori*.

Constructorul este apelat automat la instanţierea – fie statică, fie dinamică – a unei clase.

Destructorul este apelat automat la eliminarea unui obiect, adică la încheierea timpului de viaţă în cazul static, sau la apelul unui *delete* în cazul dinamic.

Din punct de vedere cronologic, constructorul este apelat după alocarea memoriei necesare, deci în faza finală a creării obiectului, iar destructorul înaintea eliberării memoriei aferente, deci în faza iniţială a distrugerii sale.

Constructorii şi destructorii se declară şi se definesc similar cu celelalte funcţii membre, dar prezintă o serie de caracteristici specifice:

- numele lor coincide cu numele clasei căreia îi aparţin; destructorii se disting de constructori prin faptul că numele lor este precedat de caracterul „~";
- nu pot returna nici un rezultat;
- nu se pot utiliza pointeri către constructori sau destructori;
- constructorii pot avea parametri, destructorii însă nu. Un constructor fără parametri poartă denumirea de *constructor implicit*.

De remarcat este faptul că în cazul în care o clasă nu dispune de constructori sau destructori, compilatorul de C++ generează automat un constructor, respectiv un destructor implicit.

Să completăm în continuare clasa *Point* cu un constructor şi un destructor:

```
Point::Point()
// constructor implicit
{
    x = 0;
    y = 0;
}

Point::Point(unsigned X, unsigned Y)
{
    x = X;
    y = Y;
}

Point::~Point()
{
}
```

Se poate observa cu această ocazie şi modul de marcare a comentariilor în C++: tot ce se află după şirul // este considerat comentariu.

De notat este faptul că definiţii de forma:

```
Point p;
```

sau

```
Point *p = new Point();
Point *p = new Point;
```

duc la apelarea constructorului implicit.

3.4. Conceptul de moştenire

Dacă întrebăm un zoolog ce este un câine, ne va răspunde că este un reprezentant al speciei *canine domesticus*. Un câine este un tip de carnivor, un carnivor este un tip de mamifer, şi aşa mai departe. Zoologul împarte animalele în regnuri, clase, ordine, familii, genuri şi specii. Această ierarhie stabileşte o relaţie de genul *„este un/este o"*. Putem remarca acest tip de relaţie oriunde în lume: Mercedes *este un* tip de maşină, care la rândul său *este un* tip de autovehicul, şi exemplele pot continua. Astfel, când spunem că ceva este un tip de altceva diferit, spunem că este o specializare a acelui lucru, aşa cum o maşină este un tip mai special de autovehicul.

Conceptul de câine *moşteneşte*, deci primeşte în mod automat, toate caracteristicile unui mamifer. Deoarece este un mamifer, cunoaştem faptul că se mişcă şi respiră aer – toate mamiferele se mişca şi respiră aer prin definiţie. Gîndindu-ne la un câine îl asociem totodată cu ideea de lătrat, cu mişcatul din coadă, şi aşa mai departe. Putem clasifica mai departe câinii în câini de pază şi câini de vânătoare, iar câinii de vânătoare, în cockeri şi dobermani etc.:

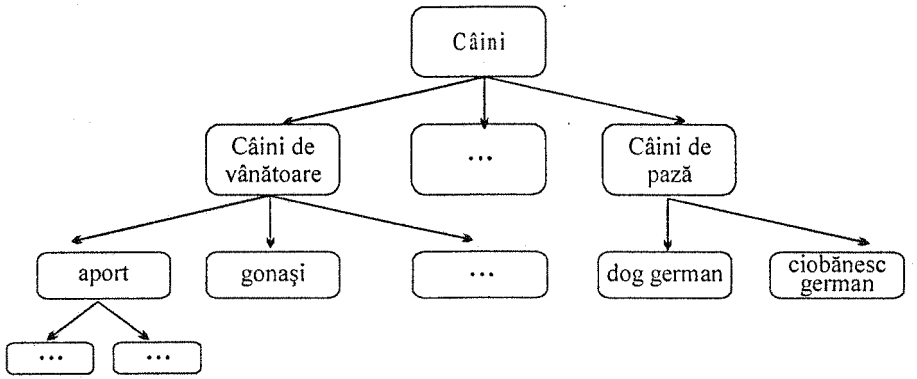


Figura 3.1. Ilustrare a conceptului de moştenire

Aşa cum am văzut mai sus, conceptul de moştenire este o noţiune foarte naturală, pe care o întâlnim în viaţa de zi cu zi. În C++ întâlnim noţiunea de *derivare*, care este în fapt o abstractizare a noţiunii de moştenire. O clasă care adaugă proprietăţi noi la o clasă deja existentă vom spune că este *derivată* din clasa originară. Clasa originară poartă denumirea de *clasă de bază*.

Clasa derivată moşteneşte toate datele şi funcţiile membre ale clasei de bază (care nu sunt declarate ca *private* în aceasta); ea poate adăuga noi date la cele existente şi poate suprascrie sau adăuga funcţii membre. Clasa de bază nu este afectată în nici un fel în urma acestui proces de derivare şi ca urmare nu trebuie recompilată. Declaraţia şi codul obiect sunt suficiente pentru crearea clasei derivate, ceea ce permite reutilizarea şi adaptarea uşoară a codului deja existent, chiar dacă fişierul-sursă nu este disponibil (spre exemplu, în cazul unei biblioteci). Astfel, nu este necesar ca programatorul unei clase derivate să cunoască modul de implementare a funcţiilor membre din componenţa clasei de bază.

O noţiune nouă legată de derivare este cea de *supraîncarcare* sau *suprascriere* a funcţiilor membre. Aceasta se referă, în mod evident, la redefinirea unor funcţii ale clasei de bază în clasa derivată. De notat este faptul că funcţiile originale din clasa-părinte sunt în continuare accesibile în clasa derivată, deci caracteristicile clasei de bază nu sunt pierdute.

Dintr-o clasă de bază pot fi derivate mai multe clase şi fiecare clasă derivată poate servi mai departe ca bază pentru alte clase derivate. Se poate astfel realiza o *ierarhie de clase*, care să modeleze adecvat sisteme complexe. Pornind de la clase simple şi generale, fiecare nivel al ierarhiei acumulează caracteristicile claselor-părinte şi le adaugă un anumit grad de specializare. Mai mult decât atât, în C++ este posibil ca o clasă derivată să moştenească simultan proprietăţile mai multor clase de bază, procedură numită *moştenire multiplă*. Construirea ierarhiei de clase reprezintă activitatea fundamentală de realizare a unei aplicaţii orientate-obiect, reprezentând în fapt faza de proiectare a respectivului sistem.

Sintaxa simplificată a derivării este:

```
class NumeClasaDerivata : NumeClasaDeBaza
```

În continuare vom deriva din clasa Point o clasă specializată, `GraphicPoint`, care va „şti" să deseneze efectiv punctul pe ecran:

```
class GraphicPoint : Point
{
    unsigned color;
    GraphicPoint(unsigned X, unsigned Y, unsigned Color);
    ~GraphicPoint();
    void Draw();
    void SetX(unsigned X);
    void SetY(unsigned Y);
};

GraphicPoint::GraphicPoint(unsigned X, unsigned Y, unsigned Color) :
    Point(X, Y)
{
    color = Color;
}

GraphicPoint::~GraphicPoint()
{
}

GraphicPoint::Draw()
{
    // ...
    // apelare primitive grafice pentru desenarea efectiva a punctului
}

GraphicPoint::SetX(unsigned X)
{
    Point::SetX(); // apelul functiei SetX() apartinand clasei de baza
    Draw();
}

GraphicPoint::SetY(unsigned Y)
{
    Point::SetY();
    Draw();
}
```


Se observă din exemplul de mai sus că am adăugat o variabilă nouă faţă de clasa `Point`, `color`, pentru a putea memora culoarea cu care se face desenarea punctului. De asemenea, am suprascris constructorul şi destructorul clasei-părinte. În constructorul derivat, am apelat constructorul original folosind construcţia:

```
ClasaDerivata::ClasaDerivata() : ClasaDeBaza()
```

În clasa `GraphicPoint` am adăugat o funcţie membră nouă, `Draw()`, care desenează efectiv punctul pe ecran. Am suprascris funcţiile `SetX()` şi `SetY()`, apelând în ambele funcţiile originare, utilizând sintaxa:

```
ClasaDeBaza::FunctieMembra()
```

apelând apoi funcţia de desenare `Draw()`.

Regulile de funcţionare a constructorilor şi destructorilor, descrise în paragraful precedent, rămân valabile şi în cazul claselor derivate, cu două observaţii privind ordinea de apelare a acestora:

- la instanţierea clasei derivate se apelează mai întâi constructorul clasei de bază, apoi se apelează propriul constructor;
- la distrugerea unui obiect al unei clase derivate, este apelat mai întâi propriul constructor, şi apoi destructorul clasei de bază (deci în ordine inversă celei de la crearea obiectului).

Capitolul 4

PROGRAMARE AVANSATĂ UTILIZÂND CLASE

Dacă un general nu este curajos,
el nu va putea să biruie incertitudinile
sau să realizeze planuri bune.
(Shen Pao-hsu)

4.1. Controlul accesului la clase

Spre deosebire de limbajele orientate-obiect pure, C++ permite controlul accesului la membrii claselor. În acest scop, s-au creat trei specificatori de control al accesului:

- `public`: membrul poate fi accesat de orice funcție din domeniul declarației clasei;
- `private`: membrul este accesibil numai funcțiilor membre și *prietene* ale clasei;
- `protected`: similar cu `private`, însă accesul se extinde și la funcțiile membre și prietene ale claselor derivate.

De remarcat este faptul că o funcție membră a unei clase are acces la toți membrii clasei, indiferent de specificatorul de acces.

Așadar, sintaxa declarației unei clase derivate incluzând controlul accesului este:

```
class NumeClasaDerivata : SpecificatorAcces NumeClasaDeBaza
```

unde *SpecificatorAcces* poate fi `public` sau `private`.

Atributul din clasa de bază	Modificator de acces	Accesul moștenit de clasa derivată	Accesul din exterior
`private` `protected` `public`	`private` `private` `private`	inaccesibil `private` `private`	inaccesibil inaccesibil inaccesibil
`private` `protected` `public`	`public` `public` `public`	inaccesibil `protected` `public`	inaccesibil inaccesibil accesibil

Se observă că pentru a oferi clasei derivate acces la un membru al clasei de bază, acesta trebuie declarat `protected` sau `public`. Pentru respectarea principiului încapsulării datelor, datele membre pentru care se oferă acces claselor derivate se declară în clasa de bază cu atributul `protected`. De asemenea, pentru a conserva dreptul de acces în urma derivării, se utilizează derivarea `public`. Accesul poate fi stopat pe orice nivel al ierarhiei de clase printr-o derivare `private`.

Stabilirea atributelor de acces ale membrilor unei clase, precum și ale derivărilor, trebuie să se facă astfel încât dezvoltarea ierarhiei de clase să nu afecteze principiul încapsulării datelor.

Vom relua în continuare exemplul din capitolul precedent, completat cu specificatori de acces:

```
class Point
{
protected:
    unsigned x, y;
public:
    Point();
    Point(unsigned X, unsigned Y);
    ~Point();
    unsigned long Arie();
    unsigned GetX();
    unsigned GetY();
    void SetX(unsigned X);
    void SetY(unsigned Y);
};

class GraphicPoint : public Point
{
    unsigned color;
    public:
       GraphicPoint(unsigned X, unsigned Y, unsigned Color);
       ~GraphicPoint();
       void Draw();
       void SetX(unsigned X);
       void SetY(unsigned Y);
};
```

Se observă că variabilele membru `x` și `y` sunt declarate `protected`, așa încât vor fi vizibile și vor avea același atribut în clasa `GraphicPoint` (deși nu sunt utilizate). În mod normal, `x` și `y` ar trebui să fie declarați `private`, întrucât nu sunt utilizați decât în interiorul clasei `Point`. Funcțiile din `GraphicPoint` nu accesează acești doi membri direct, ci prin intermediul metodelor publice de accesare oferite de clasa `Point`.

Important de notat este faptul că, implicit, dacă nu este utilizat nici un specificator de acces, membrii sunt consideraţi `private`.

```
void main()
{
    Point *p;

    p = new Point;
    p->x = 5; // operatie imposibila: x este membru privat
    p->y = 8; // operatie imposibila: y este membru privat
    p->SetX(5); /*  corect: acces la variabila x prin intermediul
                    functiei SetX() */
    p->SetY(8);
    printf("Aria = %d\n", p->Aria());
    delete p;
}
```

Am demonstrat prin acest exemplu de program că din exteriorul unei clase nu pot fi accesate datele membre `private` sau `protected`.

4.2. Funcţii şi clase prietene

În paragraful precedent, am afirmat că principiul încapsulării datelor este bine de respectat în cadrul elaborării ierarhiei de clase. Cu toate acestea, există situaţii în care este greu să se respecte acest principiu. De aceea, Bjarne Stroustrup a introdus un concept menit să rezolve şi aceste situaţii particulare, pentru a oferi soluţii elegante în vederea rezolvării tuturor situaţiilor posibile. Acest concept este cel de *friend*, care permite practic abateri controlate de la ideea protejării datelor prin încapsulare. Este recomandat ca mecanismul *friend* să fie folosit **numai** în cazul în care nu există altă soluţie!

Mecanismul *friend* (sau *prietenie*) a apărut datorită imposibilităţii de a face ca o metodă să fie membră a mai multe clase.

Funcţiile prietene sunt funcţii care nu sunt metode ale unei clase, dar care au totuşi acces la membrii privaţi ai acesteia. Orice funcţie poate fi *prietenă* a unei clase, indiferent de natura acesteia.

Sintaxa declarării unei funcţii prietene în cadrul declaraţiei unei clase este următoarea:

```
friend NumeFunctie
```

Iată şi un exemplu:

```
class Point {
    friend unsigned long Calcul(unsigned X, unsigned Y);
public:
    friend unsigned long AltaClasa::Calcul(unsigned X, unsigned Y);
    ...
};

unsigned long Calcul(unsigned X, unsigned Y)
{
    return X * Y / 2;
}

unsigned long AltaClasa::Calcul(unsigned X, unsigned Y)
{
    ...
}
```


După cum se vede din exemplul de mai sus, nu are nici o importanţă în cadrul cărei secţiuni este declarată funcţia prietenă.

Clasele prietene sunt clase care au acces la membrii privaţi ai unei clase. Sintaxa declarării unei clase prietene este:

```
friend class NumeClasaPrietena
```

Iată şi un exemplu:

```
class PrimaClasa {
    ...
};

class ADouaClasa {
    ...
    friend class PrimaClasa;
};
```

În exemplul de mai sus, clasa `PrimaClasa` are acces la membrii privaţi ai clasei `ADouaClasa`.

Important este să remarcăm că *relaţia de prietenie nu este tranzitivă*. Dacă o clasă A este prietenă a clasei B, iar clasa B este prietenă a unei clase C, aceasta nu înseamnă că A este prietenă a clasei C. De asemenea, *proprietatea de prietenie nu se moşteneşte în clasele derivate*.

4.3. Cuvântul-cheie *this*

Toate datele membre ale unei clase primesc un parametru ascuns, pointer-ul `this`, care reprezintă adresa obiectului în cauză. Acesta poate fi utilizat în cadrul funcţiilor membre.

Iată şi un exemplu:

```
unsigned long Point::Arie()
{
    return this->x * this->y;
}
```

4.4. Redefinirea operatorilor

După cum ştiţi deja, C/C++ are definite mai multe tipuri de date: *int*, *char* etc. Pentru utilizarea acestor tipuri de date sunt definiţi mai mulţi operatori: adunare (+), înmulţire (*) etc. C++ permite programatorilor să îşi redefinească (supraîncarce) aceşti operatori pentru a lucra cu propriile clase.

Sintaxa supraîncărcării unui operator este:

```
operator Simbol
```

unde *Simbol* este simbolul oricărui operator C++, exceptând: „.", „.*", „::", „?:". Această definire se face în cadrul clasei, întocmai ca pentru o funcţie membră.

Operatorii supraîncărcaţi în interiorul unei clase vor respecta reguli de asociativitate similare celor pentru operatorii predefiniţi, după cum se poate constata din tabelul de mai jos:

Tipul operatorului	Simbolul operatorului	Asociativitate	Observaţii
Binar	() [] ->	->	Se definesc doar ca funcţii membre
Unar	+ - ~ * & (tip)	<-	
Unar	++ --	<-	
Unar	new delete	<-	Poate fi supradefinit şi pentru o clasă
Binar	-> * / % + - & | && ||	->	
Binar	<< >> < <= > >= == !=	->	
Binar	= += -= *= /= %= &= ^= |= <<= >>=	<-	Se definesc doar ca funcţii membre
Binar	,	->	

După cum vom vedea în continuare, există două variante de definire a operatorilor:

- ca funcţii membre ale clasei;
- ca funcţii prietene ale clasei.

Pentru exemplificare, ne propunem să extindem clasa `Point` cu utilizarea unor operatori.

```
class Point {
    Point& operator += (Point __p);
    Point& operator -= (Point __p);
    Point operator + (Point __p);
    Point operator - (Point __p);
    Point& operator = (Point __p);
    int operator == (Point __p);
    int operator != (Point __p);
    int operator < (Point __p);
    int operator > (Point __p);
    int operator <= (Point __p);
    int operator >= (Point __p);
};

Point& Point::operator += (Point __p)
{
    x += __p.x;
    y += __p.y;
    return *this;
}

Point& Point::operator -= (Point __p)
{
    x -= __p.x;
    y -= __p.y;
    return *this;
}

Point Point::operator + (Point __p)
{
    return Point(x + __p.x, y + __p.y);
}

Point Point::operator - (Point __p)
{
    return Point(x - __p.x, y - __p.y);
}

int Point::operator == (Point __p)
{
    return x == __p.x && y == __p.y;
}

int Point::operator != (Point __p)
```

```
{
    return !(*this == __p);
}

int Point::operator < (Point __p)
{
    return x < __p.x && y < __p.y;
}

int Point::operator > (Point __p)
{
    return x > __p.x && y > __p.y;
}

int Point::operator <= (Point __p)
{
    return x <= __p.x && y <= __p.y;
}

int Point::operator >= (Point __p)
{
    return x >= __p.x && y >= __p.y;
}

main()
{
    Point p1 = Point(5, 7), p2(1, 3);

    if(p1 > p2)
      p1 -= p2;
    else
      p2 -= p1;
}
```

Am utilizat mai sus varianta cu funcţii membre. În continuare vom descrie implementarea operatorului + folosind cea de-a doua variantă, cu funcţii prietene.

```
class Point {
    // ...
    friend Point operator + (Point __p1, Point __p2);
}

Point operator + (Point __p1, Point __p2)
{
    return Point(__p1.x + __p2.x, __p1.y + __p2.y);
}
```

Definirea operatorilor ca fiind funcţii membre ale unei clase prezintă o restricţie majoră: primul operand este obligatoriu de tipul clasă respectiv.

Supradefinirea operatorilor este supusă în C++ unui set de restricţii:

- nu este permisă introducerea de noi simboluri de operatori;
- patru operatori nu pot fi redefiniţi (vezi *supra*);

- caracteristicile operatorilor nu pot fi schimbate: pluralitatea (nu se poate supradefini un operator unar ca operator binar sau invers), precedenţa şi asociativitatea;
- funcţia operator trebuie să aibă *cel puţin* un parametru de tipul clasă căruia îi este asociat operatorul supradefinit.

Programatorul are libertatea de a alege natura operaţiei realizate de un operator, însă este recomandat ca noua operaţie să fie apropiată de semnificaţia iniţială.

4.4.1. Redefinirea operatorului „="

Operatorul „=" este deja predefinit în C++, inclusiv pentru operanzi de tip clasă. Dacă nu este supradefinit, atribuirea se face membru cu membru, în mod similar cu iniţializarea obiectului efectuată de către compilator. Pot exista situaţii în care se doreşte o atribuire specifică clasei, mai ales în cazul în care clasa are date membru de tip pointer, ca atare poate fi supradefinit.

```
Point& Point::operator = (Point __p)
{
    x = __p.x;
    y = __p.y;
    return *this;
}

main()
{
    Point p1(4, 7), p2;

    p2 = p1;
}
```

4.4.2. Redefinirea operatorului *[]*

Operatorul de indexare [] se foloseşte în cazul în care există ca membru al clasei un tablou, pentru a se returna elementul de un anumit indice al acestuia, definindu-se astfel:

```
int &operator[](int)
```

Iată un exemplu de utilizare a acestui operator:

```
#define MAX 10

class SirDeIntregi
{
```

```
    int sir[MAX];
public:
    SirDeIntregi()
    {
       for(i = 0; i < MAX; i++)
          sir[i] = 0;
    }
    int &operator[](int i)
    {
       return sir[i + 1];
    }
};

main()
{
    SirDeIntregi s;

    s[1] = 2;
}
```

4.4.3. Redefinirea operatorilor *new* şi *delete*

Aceşti doi operatori pot fi supradefiniţi pentru a realiza operaţii specializate de alocare/eliberare dinamică a memoriei pentru obiectele unei clase, mai ales în cazul în care clasa are date membre de tip pointer. Funcţia operator `new` trebuie să primească un argument de tipul `size_t` care să precizeze dimensiunea în octeţi a obiectului alocat şi să returneze un pointer de tip `void` conţinând adresa zonei alocate:

```
void *operator new(size_t)
```

cu menţiunea că `size_t` este definit în `stdlib.h`. Chiar dacă parametrul de tip `size_t` este obligatoriu, calculul dimensiunii obiectului în cauză şi generarea sa se fac de către compilator.

Funcţia operator `delete` trebuie să primească ca prim parametru un pointer de tipul clasei în cauză sau `void`, conţinând adresa obiectului de distrus, şi un al doilea parametru, opţional, de tip `size_t`. Funcţia nu întoarce nici un rezultat.

```
void operator delete(void *, size_t)
```

Trebuie să menţionăm aici că operatorii `new` şi `delete` supradefiniţi păstrează toate proprietăţile operatorilor `new` şi `delete` standard.

Exemplul de mai jos implementează operatorii `new` şi `delete` folosind funcţiile clasice din C, `malloc` şi `free`:

```
#include <stdio.h>
#include <string.h>
```



```
void* operator new(size_t marime)
{
    void *nou;

    if((nou = malloc(marime)) == NULL)
       printf("Nu am putut aloca %d octeti!\n", marime);

    return nou;
}

void operator delete(void *p)
{
    free(p);
}

main()
{
    char *s;

    s = new char(255);
    strcpy(s, "test");
    delete s;
}
```

4.4.4. Redefinirea operatorilor unari

Operatorii unari pot fi supradefiniţi utilizând o funcţie membră fără parametri sau o funcţie prietenă cu un parametru de tipul clasă. Trebuie subliniat că pentru operatorii `++` şi `--` dispare distincţia dintre utilizarea ca prefix şi cea ca postfix, de exemplu între `x++` şi `++x`, respectiv `x--` şi `--x`, după cum menţionam şi în tabela cu operatori de mai sus.

4.5. Moştenirea multiplă

Limbajul C++ permite crearea de clase care moştenesc proprietăţile mai multor clase de bază. Moştenirea multiplă creşte astfel flexibilitatea dezvoltării ierarhiei de clase. Dacă derivarea normală duce la construirea unei ierarhii de tip arbore, derivarea multiplă va genera ierarhii de tip graf.

Sintaxa completă pentru operaţia de derivare este următoarea:

```
class NumeClasaDerivata : ListaClaseDeBaza
```

unde *ListaClaseDeBază* este:

```
SpecificatorAcces NumeClasaDeBaza, ...
```

4.5.1. Clase virtuale

Utilizarea moştenirii multiple se poate complica o dată cu creşterea dimensiunii ierarhiei de clase. O situaţie care poate apărea este derivarea din două clase de bază, `Clasa1` şi `Clasa2`, care la rândul lor sunt derivate dintr-o clasă comună, `ClasaDeBaza`. În acest caz, noua clasă, `ClasaNoua`, va conţine datele membre ale clasei `ClasaDeBaza` duplicate. Dacă prezenţa acestor date duplicate este utilă, ele pot fi distinse evident cu ajutorul operatorului de rezoluţie, `::`. Totuşi, în cele mai multe cazuri, această duplicare nu este necesară şi duce la consum inutil de memorie. De aceea, în C++ a fost creat un mecanism care să evite această situaţie, prin intermediul conceptului de *clasă virtuală*. Sintaxa este:

```
class NumeClasaDerivata : SpecificatorAcces virtual NumeClasaDeBaza
```

Această declaraţie nu afectează clasa în cauză, ci numai clasele derivate din aceasta. Astfel, clasele `Clasa1` şi `Clasa2` considerate vor fi declarate virtuale. Trebuie menţionat faptul că declararea `virtual` a acestor clase va afecta definirea constructorului clasei `ClasaNoua`, deoarece compilatorul nu poate hotărî care date vor fi transferate către constructorul `ClasaDeBaza`, dintre cele specificate de constructorii `Clasa1` şi `Clasa2` (putând exista, după cum am precizat, date redundante). Constructorul `ClasaNoua` va trebui modificat astfel încât să trimită datele pentru constructorul `ClasaDeBaza`. De asemenea, trebuie precizat că într-o ierarhie de clase derivate, constructorul clasei virtuale este întotdeauna apelat primul.

```
class ClasaDeTest
{
public:
    virtual bool Test(int i)
    {
       return (i == 0);
    }
};

class ClasaDerivata
{
public:
    virtual bool Test(int i)
    {
       return (i == 1);
    }
};
```



```
main()
{
    ClasaDerivata c;

    c.Test(1);
}
```

4.6. Conversii de tip definite de programator

După cum ştiţi, în C/C++ există definit un set de reguli de conversie pentru tipurile fundamentale de date. C++ permite definirea de reguli de conversie pentru clasele create de programator. Regulile astfel definite sunt supuse unor restricţii:

- într-un şir de conversii nu este admisă decât o singură conversie definită de programator;
- se recurge la aceste conversii numai după ce se verifică existenţa altor soluţii (de exemplu, pentru o atribuire, se verifică mai întâi supraîncărcarea operatorului de atribuire şi în lipsa acestuia se face conversia).

Există două metode de a realiza conversii de tip, pe care le vom prezenta în cele ce urmează.

4.6.1. Supraîncărcarea operatorului unar *cast*

Sintaxa este:

```
operator TipData()
```

respectiv:

```
operator (TipData)
```

Operatorul *cast* este unar, aşadar are un singur operand, adresa obiectului prin intermediul căruia este apelat, şi întoarce un rezultat de tipul operatorului. Ca urmare, prin această metodă se pot defini numai conversii dintr-un tip clasă într-un tip de bază sau un alt tip clasă.

De remarcat este faptul că, în cazul conversiei dintr-un tip clasă într-un alt tip clasă, funcţia operator trebuie să aibă acces la datele membre ale clasei de la care se face conversia, deci trebuie declarată prietenă a clasei respective.

Exemplul de mai jos implementează un operator care converteşte un obiect de tip `Point` într-un întreg:

```
class Point
{
    int x, y;
```

```
public:
    // transforma un Point intr-un int
    operator int ()
    {
       return x * y;
    }
};

main()
{
    Point p(10, 72);

    printf("Arie: %d\n", (int)p);
}
```

4.6.2. Conversii de tip folosind constructori

Aceasta metodă constă în definirea unui constructor ce primeşte ca parametru tipul de la care se face conversia. Constructorul întoarce întotdeauna ca rezultat un obiect de tipul clasei de care aparţine. Ca urmare, folosind aceasta metodă se pot realiza numai conversii dintr-un tip de bază sau un tip clasă într-un tip clasă.

Trebuie menţionat faptul că, în cazul conversiei dintr-un tip clasă într-un alt tip clasă, constructorul trebuie să aibă acces la datele membre ale clasei de la care se face conversia, deci trebuie declarată prietenă a clasei respective.

```
class CuloareRGB
{
    int rosu, verde, albastru;
public:
    CuloareRGB(int c_rosu, int c_verde, int c_albastru)
    {
       rosu = c_rosu;
       verde = c_verde;
       albastru = c_albastru;
    }
    CuloareRGB()
    {
       rosu = verde = albastru = 0;
    }
};

class Culoare
{
    int culoare;
public:
    // transforma un tip CuloareRGB in Culoare
    Culoare(CuloareRGB rgb)
    {
       return rgb.rosu * 256 + rgb.verde * 16 + rgb.albastru;
    }
};
```


4.7. Constructorul de copiere

O situaţie care poate apărea deseori este iniţializarea unui obiect cu datele membre ale unui obiect de acelaşi tip. În paragraful 4.4.1 am descris supraîncărcarea operatorului de atribuire. Există totuşi situaţii în care acest operator nu poate fi utilizat, cum ar fi la transferul unui obiect ca parametru sau la crearea unei instanţe temporare a unei clase, atunci când copierea membru cu membru nu este adecvată (de exemplu, datorită existenţei unor membri de tip pointer). Pentru a rezolva aceste situaţii, C++ a introdus un constructor special, numit *constructorul de copiere*.

Sintaxa este:

```
NumeClasa::NumeClasa (NumeClasa &NumeObiectSursa)
```

În continuare vom completa clasa `Point` cu un constructor de copiere:

```
Point::Point(Point &p)
{
    x = p.x;
    y = p.y;
}
```

În cazul în care clasa nu dispune de constructor de copiere, compilatorul generează automat un constructor de copiere care realizează copierea membru cu membru.

4.8. Clase abstracte

În C++ există posibilitatea de a defini clase generale, care sunt destinate exclusiv creării de noi clase prin derivare, ele neputând fi instanţiate şi utilizate ca atare (pot fi declaraţi doar pointeri către astfel de clase, care vor fi iniţializaţi însă cu adresele unor obiecte ale claselor derivate). Clasele de acest gen se numesc *clase abstracte*. Ele se constituie ca bază în cadrul elaborării de ierarhii de clase, putând fi folosite, spre exemplu, pentru a impune anumite restricţii în realizarea claselor derivate.

În vederea construirii unor astfel de clase, s-a introdus conceptul de *funcţie virtuală pură*. O astfel de funcţie este declarată în cadrul clasei, dar nu este definită. O clasă care conţine o funcţie virtuală pură este considerată abstractă. Sintaxa definirii acestor funcţii este:

```
virtual TipData NumeFunctieMembra() = 0
```

Se impune aici observaţia că funcţiile virtuale pure trebuie redefinite în clasele derivate, altfel şi acestea vor fi considerate abstracte.

```
// clasa abstracta
class XEventListener
{
    public:
    // functie virtuala pura
    bool ProcessEvent(XEvent *__Event) = 0;
};

class XObjectEventListener : public XEventListener
{
 public:
    virtual bool ProcessEvent(XEvent *__Event);
    virtual bool ProcessMouseEvent(XMouseEvent *__Event);
    virtual bool ProcessKeyboardEvent(XKeyboardEvent *__Event);
    virtual bool ProcessMessage(XMessage *__Event);
    virtual bool ProcessCommand(XMessage *__Event);
    virtual bool ProcessSignal(XEvent *__Event);
    virtual bool ProcessBroadcast(XEvent *__Event);
};
```

4.9. Membri statici ai unei clase

În mod normal, datele membre ale unei clase sunt alocate în cadrul fiecărui obiect. În C++, se pot defini date membre cu o comportare specială, numite *date statice*. Acestea sunt alocate o singură dată, existând sub forma unei singure copii, comună tuturor obiectelor de tipul clasă respectiv, iar operaţiunile de creare, iniţializare şi acces la aceste date sunt independente de obiectele clasei. Sintaxa este:

```
static DeclarareMembru
```

Funcţiile membre statice efectuează de asemenea operaţii care nu sunt asociate obiectelor individuale, ci întregii clase (de aceea, nu pot accesa decât datele membre statice ale clasei, nu şi pe cele obişnuite). Funcţiile exterioare clasei pot accesa membrii statici ai acesteia astfel:

```
NumeClasa::NumeMembru

Obiect::NumeMembru
```


Funcţiile membre statice nu primesc ca parametru implicit adresa unui obiect, aşadar în cadrul lor cuvântul-cheie `this` nu poate fi utilizat. De asemenea, membrii normali ai clasei nu pot fi referiţi în exterior decât specificând numele unui obiect, nu doar prin intermediul numelui clasei.

```
class ClasaDeTest
{
    // o variabila membra statica
    static int rezultat;
};
```

Capitolul 5

FLUXURI

Feriți-vă de bug-urile din programul următor;
eu doar am demonstrat că este corect, nu l-am încercat.
(Donald Knuth)

5.1. Introducere

Fluxurile (*streams*) au în principal rolul de a abstractiza operațiile de intrare/ieşire. Ele oferă metode de scriere şi citire a datelor independente de dispozitivul I/O şi chiar independente de platformă. Fluxurile încapsulează (ascund) problemele specifice dispozitivului cu care se lucrează, sub biblioteca standard `iostream`.

Alt avantaj al folosirii fluxurilor este datorat implementării bibliotecii `iostream`, care utilizează un sistem de buffer-e. Se ştie că în general operațiile de intrare/ieşire care utilizează dispozitivele periferice sunt relativ mari consumatoare de timp, astfel încât aplicațiile sunt uneori nevoite să aştepte terminarea acestor operațiuni. Informațiile trimise către un flux nu sunt remise imediat dispozitivului în cauză, ci sunt transferate într-o zonă de memorie tampon, din care sunt descărcate către dispozitiv abia în momentul umplerii acestei zone de memorie.

În C++, fluxurile au fost implementate utilizând clase, după cum urmează:

- clasa `streambuf` gestionează buffer-ele;
- clasa `ios` este clasa de bază pentru clasele de fluxuri de intrare şi de ieşire. Clasa `ios` are ca variabilă membru un obiect de tip `streambuf`;
- clasele `istream` şi `ostream` sunt derivate din `ios`;
- clasa `iostream` este derivată din `istream` şi `ostream` şi oferă metode pentru lucrul cu terminalul;
- clasa `fstream` oferă metode pentru operații cu fişiere.

5.2. Obiecte standard

Când un program C++ care include `iostream.h` este lansat în execuție, sunt create şi inițializate automat patru obiecte:

- `cin` gestionează intrarea de la intrarea standard (tastatura);
- `cout` gestionează ieşirea către ieşirea standard (ecranul);
- `cerr` gestionează ieşirea către dispozitivul standard de eroare (ecranul), neutilizând buffer-e;
- `clog` gestionează ieşirea către dispozitivul standard de eroare (ecranul), utilizând buffer-e.

5.3. Redirectări

Dispozitivele standard de intrare, ieşire şi eroare pot fi *redirectate* către alte dispozitive. Erorile sunt de obicei redirectate către fişiere, iar intrarea şi ieşirea pot fi *conduse* (engl. *piped*) către fişiere utilizând comenzi ale sistemului de operare (utilizarea ieşirii unui program ca intrare pentru altul).

Sintaxa pentru operații de ieşire, `cout`:

```
cout << InformatieDeTrimisLaIesire;
```

respectiv pentru intrare, `cin`:

```
cin >> NumeVariabila;
```

De fapt, `cin` şi `cout` sunt nişte obiecte definite global, care au supraîncărcat operatorul >> respectiv << de mai multe ori, pentru fiecare tip de parametru în parte (`int`, `char *` etc.):

```
istream &operator >> (TipParametru &)
```

De exemplu:

```
#include <iostream.h>

void main()
{
```

```
    int NumarIntreg;

    cout << "NumarIntreg = ";
    cin >> NumarIntreg;
    cout << "\nCe ai introdus = " << NumarIntreg << endl;
}
```

Acest scurt program citeşte de la intrarea standard o valoare întreagă, pe care o trimite apoi către ieşirea standard. Se observă posibilitatea de a utiliza simbolurile „\n", „\t", ş.a.m.d. (ca la `printf`, `scanf` etc.). Utilizarea simbolului `endl` va forţa golirea zonei tampon, adică trimiterea datelor imediat către ieşire.

Atât operatorul >>, cât şi << returnează o referinţă către un obiect al clasei `istream`. Deoarece `cin`, respectiv `cout` sunt şi ele obiecte `istream`, valoarea returnată de o operaţie de citire/scriere din/în `stream` poate fi utilizată ca intrare/ieşire pentru următoarea operaţie de acelaşi fel.

5.4. *cin*

Funcţia `cin.get()`

Funcţia membră `get()` poate fi utilizată pentru a obţine un singur caracter din intrare, apelând-o fără nici un parametru, caz în care returnează valoarea utilizată, sau cu un parametru de tip pointer la tipul `char`.

- **`get()` fără parametri**

În această formă, funcţia întoarce valoarea caracterului găsit. De remarcat este faptul că, spre deosebire de operatorul >>, nu poate fi utilizată pentru a citi mai multe intrări, deoarece valoarea returnată este de tip întreg, nu un obiect `istream`. Mai jos, un exemplu de utilizare:

```
#include <iostream.h>

void main()
{
    char c;

    while((c = cin.get()) != EOF)
    {
       cout << "c = " << c << endl;
    }
}
```

BIBL. M. Fluxuri CENTRU UNIV.

Citirea de şiruri de caractere utilizând `get()`

Operatorul >> nu poate fi utilizat pentru a citi corect şiruri de caractere de la intrare deoarece spaţiile sunt interpretate ca separator între diverse valori de intrare. În astfel de cazuri trebuie folosită funcţia `get()`. Sintaxa de utilizare a funcţiei `get`, în acest caz, este următoarea:

```
cin.get(char *PointerLaSirulDeCaractere, int LungimeMaxima, char
            Sfarsit);
```

Primul parametru este un pointer la zona de memorie în care va fi depus şirul de caractere. Al doilea parametru reprezintă numărul maxim de caractere ce poate fi citit plus unu. Cel de-al treilea parametru este caracterul de încheiere a citirii, care este opţional (implicit considerat „\n"). În cazul în care caracterul de încheiere este întâlnit înainte de a fi citit numărul maxim de caractere, acest caracter nu va fi extras din flux. Există o funcţie similară funcţiei `get()`, cu aceeaşi sintaxă, numită `getline()`. Funcţionarea sa este identică cu `get()`, exceptând faptul că acel ultim caracter menţionat mai sus este şi el extras din flux.

Funcţia `cin.ignore()`

Această funcţie se utilizează pentru a trece peste un număr de caractere până la întâlnirea unui anumit caracter. Sintaxa sa este:

```
cin.ignore(int NumarMaximDeCaractere, char Sfarsit);
```

Primul parametru reprezintă numărul maxim de caractere ce vor fi ignorate, iar al doilea parametru, caracterul care trebuie găsit.

Funcţia `cin.peek()`

Această funcţie returnează următorul caracter din flux, fără însă a-l extrage.

Funcţia `cin.putback()`

Această funcţie inserează în flux un caracter.

5.5. *cout*

5.5.1. Funcţii membre ale *cout*

Funcţia `cout.flush()`

Funcţia `cout.flush()` determină trimiterea către ieşire a tuturor informaţiilor aflate în zona de memorie tampon. Această funcţie poate fi apelată şi în forma `cout << flush`.

Funcţia `cout.put()`

Funcţia `cout.put()` scrie un caracter către ieşire. Sintaxa sa este următoarea:

```
cout.put(char Caracter);
```

Deoarece această funcţie returnează o referinţă de tip `ostream`, pot fi utilizate apeluri succesive ale acesteia, ca în exemplul de mai jos:

```
#include <iostream.h>

void main()
{
    cout.put('H').put('i').put('!').put('\n');
}
```

Funcţia `cout.write()`

Această funcţie are acelaşi rol ca şi operatorul <<, cu excepţia faptului că se poate specifica numărul maxim de caractere ce se doresc scrise. Sintaxa funcţiei `cout.write()` este:

```
cout.write(char *SirDeCaractere, int CaractereDeScris);
```

5.5.2. Formatarea ieşirii

Funcţia cout.width()

Această funcţie permite modificarea dimensiunii valorii trimise spre ieşire, care este considerată implicit exact mărimea câmpului în cauză. Ea modifică dimensiunea numai pentru următoarea operaţie de ieşire. Sintaxa este:


```
cout.width(int Dimensiune);
```

Funcţia `cout.fill()`

Funcţia `cout.fill()` permite modificarea caracterului utilizat pentru umplerea eventualului spaţiu liber creat prin specificarea unei dimensiuni mai mari decât cea necesară ieşirii, în funcţia `cout.width()`. Sintaxa acesteia este:

```
cout.fill(char Caracter);
```

5.5.3. Opţiuni de formatare a ieşirii

Pentru formatarea ieşirii sunt definite două funcţii membre ale `cout`, şi anume:

Funcţia `cout.setf()`

Această funcţie activează o opţiune de formatare a ieşirii, primită ca parametru:

```
cout.setf(ios::Optiune);
```

unde *Optiune* poate fi:
- `showpos` determină adăugarea semnului plus (+) în faţa valorilor numerice pozitive;
- `left`, `right`, `internal` schimbă alinierea ieşirii;
- `dec`, `oct`, `hex` schimbă baza de numeraţie pentru valori numerice;
- `showbase` determină adăugarea identificatorului bazei de numeraţie în faţa valorilor numerice.

Funcţia `cout.setw()`

Această funcţie modifică dimensiunea ieşirii, fiind similară funcţiei `cout.width()`. Sintaxa sa este:

```
cout.setw(int Dimensiune);
```

În continuare vom exemplifica utilizarea funcţiilor pentru formatarea ieşirii:

```
#include <iostream.h>
#include <iomanip.h>

void main()
{
    int number = 783;
```

```
    cout << "number = " << number;
    cout.setf(ios::showbase);
    cout << "number in hexa = " << hex << number;
    cout.setf(ios::left);
    cout << "number in octal, aligned to the left = " << oct <<
       number;
}
```

5.6. Operaţii de intrare/ieşire cu fişiere

Operaţiile asupra fişierelor se efectuează prin intermediul clasei `ifstream` pentru citire, respectiv `ofstream` pentru scriere. Pentru a le utiliza, aplicaţiile trebuie să includă `fstream.h`. Clasele `ofstream` şi `ifstream` sunt derivate din clasa `iostream`, ca urmare toţi operatorii şi toate funcţiile descrise mai sus sunt moştenite şi de această clasă.

Sintaxele pentru constructorii acestor două clase sunt:

```
ofstream Variabila(char *NumeFisier, ios::Mod);

ifstream Variabila(char *NumeFisier);
```

Aceşti constructori au rolul de a deschide fişierul specificat ca parametru. Cel de-al doilea parametru al constructorului `ofstream` este opţional şi specifică modul de deschidere a fişierului:

- `ios::append` – adaugă la sfârşitul fişierului;
- `ios::atend` – poziţionează pointer-ul la sfârşitul fişierului, însă informaţiile pot fi scrise oriunde în cadrul fişierului;
- `ios::truncate` – este modul de deschidere implicit: vechiul conţinut al fişierului este pierdut;
- `ios::nocreate` – dacă fişierul nu există, atunci operaţia eşuează;
- `ios::noreplace` – dacă fişierul există deja, atunci operaţia eşuează.

Pot fi utilizate prescurtările `app` pentru `append`, `ate` pentru `atend` şi `trunc` pentru `truncate`.

Pentru a închide aceste fişiere trebuie apelată funcţia membră `close()`.

Rezultatul operaţiilor de intrare/ieşire poate fi testat prin intermediul a patru funcţii membre:

- `eof()` verifică dacă s-a ajuns la sfârşitul fişierului;
- `bad()` verifică dacă s-a executat o operaţie invalidă;
- `fail()` verifică dacă ultima operaţie a eşuat;
- `good()` verifică dacă toate cele trei rezultate precedente sunt false.

Capitolul 6

TRATAREA EXCEPŢIILOR

Isaac Newton a recunoscut în secret
câtorva prieteni: el înţelegea
comportamentul gravitaţiei,
dar nu şi modul de funcţionare.
(Lily Tomlin, *The Search for Signs of Intelligent Life in the Universe*)

Este un fenomen „natural" ca în programe să se strecoare erori, de diverse naturi. Activitatea de programare implică şi acţiuni mai puţin plăcute, adică testarea, depanarea şi corectarea erorilor. Costurile de îndepărtare a erorilor cresc de obicei direct proporţional cu întârzierea momentului din cadrul procesului de dezvoltare când sunt descoperite.

Trebuie însă înţeleasă diferenţa dintre erori (bug-uri) şi excepţii. Excepţiile sunt situaţiile neaşteptate apărute în cadrul sistemului care rulează un program. Programele trebuie să fie pregătite pentru a trata aceste situaţii excepţionale.

În C++ s-a realizat un mecanism facil de tratare a excepţiilor. Astfel, o excepţie este un obiect a cărui adresă este trimisă dinspre zona de cod unde a apărut problema către o zonă de cod care trebuie să o rezolve.

Paşii care trebuie în general urmaţi în vederea tratării excepţiilor în cadrul programelor C++ sunt:

1. Se identifică acele zone din program în care se efectuează o operaţie despre care se cunoaşte că ar putea genera o excepţie, şi se marchează în cadrul unui bloc de tip `try`. În cadrul acestui bloc, se testează condiţia de apariţie a excepţiei şi, în caz pozitiv, se semnalează apariţia excepţiei prin intermediul cuvântului-cheie `throw`.
2. Se realizează blocuri de tip `catch` pentru a capta excepţiile atunci când acestea sunt întâlnite.
3. Blocurile `catch` urmează un bloc `try`, şi în cadrul lor sunt tratate excepţiile.

Sintaxa pentru `try`:

```
try
{
    // cod
    throw TipExceptie;
}
```

Sintaxa pentru `throw`:

```
throw TipExceptie;
```

Sintaxa pentru `catch`:

```
catch(TipExceptie)
{
    // cod tratare exceptie
}
```

Dacă *TipExceptie* este „. . .", este captată orice excepţie apărută.

După un bloc `try`, pot urma unul sau mai multe blocuri `catch`. Dacă excepţia corespunde cu una dintre declaraţiile de tratare a excepţiilor, aceasta este apelată. Dacă nu este definită nici o rutină de tratare a excepţiei, este apelată rutina predefinită, care încheie execuţia programului în curs. După ce rutina este executată, programul continuă cu instrucţiunea imediat următoare blocului `try`.

TipExceptie nu este altceva decât instanţierea unei clase vide (care determină tipul excepţiei), putând fi declarat ca:

```
class TipExceptie {};
```

În continuare prezentăm un exemplu de program care utilizează tratarea excepţiilor.

```
#include <iostream.h>

#define MAXX  80
#define MAXY  25

class Point
{
public:

    class xZero {};
    class xOutOfScreenBounds {};

    Point(unsigned __x, unsigned __y)
    {
       x = __x;
       y = __y;
    }

    unsigned GetX()
    {
       return x;
    }

    unsigned GetY()
    {
       return y;
    }

    void SetX(unsigned __x)
    {
       if(__x > 0)
          if(__x <= MAXX)
             x = __x;
          else
             throw xOutOfScreenBounds();
       else
          throw xZero();
    }

    void SetY(unsigned __y)
    {
       if(__y > 0)
          if(__y <= MAXY)
             y = __y;
          else
             throw xOutOfScreenBounds();
       else
          throw xZero();
    }

protected:
    int x, y;
};

main()
{
    Point p(1, 1);

    try
    {
       p.SetX(5);
       cout << "p.x successfully set to " << p.GetX() << "." << endl;
       p.SetX(100);
    }

    catch(Point::xZero)
    {
       cout << "Valoare zero!\n";
    }

    catch(Point::xOutOfScreenBounds)
    {
       cout << "Out of screen bounds!\n";
    }

    catch(...)
    {
       cout << Exceptie necunoscuta!\n";
    }
}
```


Datorită faptului că excepţia este instanţierea unei clase, prin derivare pot fi realizate adevărate ierarhii de tratare a excepţiilor. Trebuie avută însă în vedere posibilitatea de apariţie a unor excepţii chiar în cadrul codului de tratare a unei excepţii, situaţii care trebuie evitate.

Capitolul 7

TEMPLATE-URI

Aceia dintre noi care suntem arhitecți avem poate această dorință aproape de centrul vieții noastre: că într-o zi, undeva, cumva, vom construi o clădire care este extraordinară, frumoasă, un loc în care oamenii pot merge și visa timp de secole.

(Christopher Alexander)

Template-ul implementează așa-zisul concept de *tip parametrizat* (engl. *parametrized type*). Un template reprezintă o familie de tipuri sau funcții, cu alte cuvinte, un șablon sau model. Acest concept a fost introdus în primul rând pentru a crește gradul de reutilizabilitate a codului. De exemplu, pentru a implementa o listă de numere întregi este necesară în mod normal realizarea unei clase speciale (să spunem `ListOfIntegers`), iar pentru o listă de șiruri, altă clasă (să spunem `ListOfStrings`). Conceptul de *template* permite realizarea unei clase generale (să spunem `List`), care să accepte orice tip de element, inclusiv tipuri necunoscute la momentul implementării acesteia. Tipul template-ului este stabilit în momentul instanțierii sale. Template-urile sunt foarte utile pentru realizarea de biblioteci care trebuie să ofere metode generice de prelucrare a datelor.

Sintaxa generală de declarare a unui template este următoarea:

```
template < ListaDeParametri > Declaratie
```

unde *Declaratie* reprezintă declararea sau definirea unei clase sau funcții, definirea unui membru static al unei clase template, definirea unei clase sau funcții membre a unei clase template, sau definirea unui membru template al unei clase.

Clasele parametrizate (sau clasele template) se declară astfel:

```
template <class NumeParametru>
class NumeClasa
{
    // ...
    // definirea clasei
}
```


Particularizarea (= stabilirea tipului) clasei template se face prin intermediul unei construcții de genul:

```
NumeClasa <NumeParametru>
```

unde *NumeParametru* reprezintă tipul obiectului.

Funcțiile template se declară astfel:

```
template <class NumeParametru>
// ...
// declaratia functiei
```

Să considerăm în continuare ca exemplu implementarea unei stive generice folosind template-uri.

```
#include <iostream.h>

template <class T> class StackItem
{
public:
    StackItem *Next;
    T *Data;
    StackItem(T __Data, StackItem <T> *__Next)
    {
       Data = new T(__Data);
       Next = __Next;
    }
};

template <class T> class Stack
{
public:
    T pop()
    {
       T result = *(Data->Data);
       StackItem <T> *temp = Data;
       Data = Data->Next;
       delete temp;
       return result;
    }

    T top()
    {
       return *(Data->Data);
    }

    void push(T __Data)
    {
       Data = new StackItem <T>(__Data, Data);
    }
```

```
    int isEmpty()
    {
       return Data == 0;
    }

    Stack()
    {
       Data = 0;
    }

private:
    StackItem <T> *Data;
};

main()
{
    Stack <int> anIntegerStack;
    anIntegerStack.push(5);
    anIntegerStack.push(7);
    if(anIntegerStack.isEmpty())
       cout << "Stiva goala" << endl;
    else
       cout << anIntegerStack.pop() << endl;
    cout << anIntegerStack.top() << endl;
}
```

În exemplul următor a fost implementată o listă generică (`List`). Ca elemente ale listei s-au folosit obiecte de tip `Point` (clasa definită în capitolul 4). Pentru parcurgerea uşoară a listei, a fost implementată o clasă de tip *iterator*, care poate fi considerată ca fiind un „cursor" care străbate lista. Funcţia `List.begin()` returnează un iterator poziţionat pe primul element al listei, `List.end()` pe ultimul element al listei. Saltul la următorul element al listei se face cu ajutorul operatorului ++ din clasa `Iterator`.

```
#include <iostream.h>

class Point
{
    friend ostream& operator << (ostream& output, Point p);

protected:
    unsigned x, y;

public:
    Point()
    {
       x = 0;
       y = 0;
    }

    Point(unsigned X, unsigned Y)
```



```
    {
       x = X;
       y = Y;
    }

    ~Point() {}

    unsigned GetX()
    {
       return x;
    }

    unsigned GetY()
    {
       return y;
    }

    void SetX(unsigned X)
    {
       x = X;
    }

    void SetY(unsigned Y)
    {
       y = Y;
    }
};

ostream& operator << (ostream& output, Point p)
{
    output << "(" << p.x << ", " << p.y << ").";
    return output;
}

template <class T> class Item {
public:
    Item *Next;
    T *Data;
    Item(T __Data, Item <T> *__Next)
    {
       Data = new T(__Data);
       Next = __Next;
    }
};

template <class T> class List {
public:
    T pop_front()
    {
       T result = *(Data->Data);
       Item <T> *temp = Data;
       Data = Data->Next;
       delete temp;
       return result;
    }
```

```
    T front()
    {
       return *(Data->Data);
    }

    void push_front(T __Data)
    {
       Data = new Item <T>(__Data, Data);
    }

    int empty()
    {
       return Data == 0;
    }

    List()
    {
       Data = 0;
    }

    class Iterator
    {
       friend class List <T>;

    protected:
       Item <T> *Current;

       Iterator(Item <T> *x)
       {
          Current = x;
       }

    public:
       Iterator() {}

       int operator == (Iterator& x)
       {
          return Current == x.Current;
       }

       int operator != (Iterator& x)
       {
          return Current != x.Current;
       }

       T operator *()
       {
          return *(Current->Data);
       }

       Iterator& operator ++(int)
       {
          Current = Current->Next;
          return *this;
       }
    };
```



```
    Iterator begin()
    {
       return Iterator(Data);
    }

    Iterator end()
    {
       Item <T> *temp;
       for(temp = Data; temp; temp = temp->Next);
       return Iterator(temp);
    }

private:
    Item <T> *Data;
};

main()
{
    List <Point> anPointList;
    List <Point>::Iterator index, end;

    anPointList.push_front(Point(1, 1));
    anPointList.push_front(Point(3, 14));
    index = anPointList.begin();
    end = anPointList.end();
    if(anPointList.empty())
       cout << "Lista vida" << endl;
    else
       for(; index != end; index++)
          cout << *index << " ";
    cout << endl;
}
```

Clasele template pot avea trei tipuri de *prieteni* (*friends*):

- o clasă sau o funcţie care nu este de tip template;
- o clasă sau o funcţie template;
- o clasă sau o funcţie template având tipul specificat.

Dacă este necesară particularizarea unei funcţii template sau a unei funcţii membre a unei clase template pentru un anumit tip, funcţia respectivă poate fi supraîncărcată pentru tipul dorit.

Trebuie remarcat de asemenea că în cazul în care o clasă template conţine membri statici, fiecare instanţă a template-ului în cauză va conţine propriile date statice.

În cele ce urmează vom prezenta o clasă utilizată pentru conectarea la un server SQL, în cazul nostru PostgreSQL, precum şi un scurt program care o utilizează. Se remarcă comentariile în stil DOC++ în fişierul-antet `toolbox.h`, care vor ajuta la generarea automată a documentaţiei aferente.

toolbox.h:

```
#ifndef _TOOLBOX_H
#define _TOOLBOX_H

#include <libpq-fe.h>
#include <libpq-int.h>

/** Returneaza timpul exprimat in secunde, pornind de la reprezentarea
    tip AAAA-LL-ZZ
*/
unsigned long get_time(const char *__s);

/// Returneaza timpul exprimat in secunde
unsigned long get_time(unsigned long __year,
    unsigned long __month,
    unsigned long __day);

/// Extrage anul
int get_year(unsigned long __time);

/// Extrage luna
int get_month(unsigned long __time);

/// Extrage ziua
int get_day(unsigned long __time);

/// Extrage ora
int get_hour(unsigned long __time);

/// Extrage minutul
int get_minute(unsigned long __time);

/// Extrage secunda
int get_second(unsigned long __time);

/// Returneaza o valoare de tip caracter
char get_char(const char *__s);

/// Returneaza o valoare de tip bool
bool get_bool(const char *__s);

/// Returneaza o valoare de tip long
long get_long(const char *__s);

/// Returneaza o valoare de tip double
double get_double(const char *__s);

/// Returneaza o valoare de tip sir de caractere
char *get_string(char *__s);

/// clasa care descrie o conexiune la o baza de date
```



```
class SqlDatabase {
public:
    /// face conectarea la baza de date specificata
    SqlDatabase(const char *__name);

    /// inchide conexiunea la baza de date
    ~SqlDatabase();

    /// trimite serverului o interogare SQL
    ExecStatusType Exec(const char *__query);

    /**trimite serverului o interogare SQL si returneaza 1
       daca aceasta a fost executata cu succes
    */
    int ExecCommandOk(const char *__query);

    /**trimite serverului o interogare SQL care extrage date
       dintr-o baza de date si returneaza 1 daca aceasta
       a fost executata cu succes
    */
    int ExecTuplesOk(const char *__query);

    /** Returneaza mesajul de eroare generat de ultima interogare
    */
    const char *ErrorMessage();

    /**Returneaza numarul de tuple ("inregistrari") extrase la
       ultima interogare
    */
    int Tuples();

    /** Returneaza numarul de campuri extrase la ultima interogare
    */
    int Fields();

    /// Returneaza numele unui camp
    const char *FieldName(int __field_num);

    /// Returneaza identificatorul unui camp
    int FieldNum(const char *__field_name);

    /// Returneaza tipul unui camp
    Oid FieldType(int __field_num);

    /// Returneaza tipul unui camp
    Oid FieldType(const char *__field_name);

    /// Returneaza lungimea unui camp
    int FieldSize(int __field_num);

    /// Returneaza lungimea unui camp
    int FieldSize(const char *__field_name);

    /** Returneaza valoarea unui camp dintr-o inregistrare specificata
    */
```

```
    const char *GetValue(int __tup_num, int __field_num);

    /** Returneaza valoarea unui camp dintr-o inregistrare specificata
    */
    const char *GetValue(int __tup_num,
       const char *__field_name);

    /** Returneaza lungimea unui camp dintr-o inregistrare specificata
    */
    int GetLength(int __tup_num, int __field_num);

    /** Returneaza lungimea unui camp dintr-o inregistrare specificata
    */
    int GetLength(int __tup_num, const char *__field_name);

    /// Elibereaza memoria alocata pentru inregistrari
    void Clear();

    /// Verifica daca o conexiune a esuat
    int ConnectionBad();

    /// Returneaza starea ultimei interogari
    unsigned long OidStatus();

    /// Returneaza starea conexiunii
    ConnStatusType Status();
 protected:
    PGconn *pgConn;
    PGresult *pgResult;
    ExecStatusType pgStatus;
};

void log(char *__fmt, ...);

extern char *username;

#endif
```

toolbox.cc:

```
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <syslog.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#include "toolbox.h"

char *username;
```



```
int get_year(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_year + 1900;

    return result;
}

int get_month(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_mon + 1;

    return result;
}

int get_day(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_mday;

    return result;
}

int get_hour(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_hour;

    return result;
}

int get_minute(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_min;

    return result;
}

int get_second(unsigned long __time)
{
    struct tm *t = localtime((const time_t *)&__time);
    int result = t->tm_sec;

    return result;
}

char get_char(const char *__s)
{
    return __s[0];
}

bool get_bool(const char *__s)
```

1.2. Alte editoare: *mcedit, joe* şi *vi*

1.2.1. Programul *mcedit*

Programul `mcedit` este editorul din cadrul managerului de fişiere `mc`. Este un editor simplu, oferind o serie de comenzi de bază, util fiind mecanismul de *syntax highlighting*, adică diferite scheme de colorare a instrucţiunilor, parantezelor etc. Există şi posibilitatea de a defini macrouri şi de a emula comenzile `Emacs`.

Acest editor face parte din pachetul `mc`. Apelarea lui se poate face fie prin lansarea comenzii `mcedit`, fie din cadrul `MC` cu tasta `F4` sau `Shift-F4`.

Comenzile principale sunt:

- `F1` apelează pagina de ajutor
- `F2` salvează conţinutul fişierului
- `F3` marchează începutul, respectiv sfârşitul unui bloc
- `F4` caută şi înlocuieşte un şir de caractere
- `F5` copie conţinutul blocului
- `F6` mută conţinutul blocului
- `F7` caută un şir de caractere
- `F8` şterge conţinutul blocului
- `F9` apelează meniul `mcedit`
- `F10` iese din editor
- `Ctrl-sageti stanga/dreapta` salt la cuvântul precedent/următor
- `Shift-sageti` selectează un bloc
- `Ctrl-Insert` copie blocul selectat în buffer (*clipboard*)
- `Shift-Insert` inserează conţinutul din buffer
- `Meta-L` salt la o anumită linie

Pentru semnificaţia tastei `Meta`, a se consulta paragraful despre `Emacs`.

1.2.2. Programul *joe*

Programul `joe` este un editor minimal care permite următoarele comenzi:

- `Ctrl-Z` salt la cuvântul următor
- `Ctrl-X` salt la cuvântul precedent
- `Ctrl-K F` căutare în text
- `Ctrl-L` continuarea căutării

- `Ctrl-K L` salt la o anumită linie
- `Ctrl-K B` marchează începutul blocului
- `Ctrl-K K` marchează sfârşitul blocului
- `Ctrl-K M` mută conţinutul blocului
- `Ctrl-K C` copie conţinutul blocului
- `Ctrl-K Y` şterge conţinutul blocului
- `Ctrl-Y` şterge linia curentă
- `Ctrl-K X` salvează fişierul şi părăseşte editorul

1.2.3. Programul *vi*

`vi` este un editor de texte care se găseşte în majoritatea sistemelor UNIX. Există peste 20 de implementări de `vi`. Prima versiune a fost dezvoltată la Universitatea Berkeley în anul 1980.

În Linux este răspândită implementarea numită `VIM` (*Vi IMproved*). Pentru a avea acces la editorul `vim`, trebuie instalate pachetele `vim-common`, `vim-minimal` şi eventual `vim-enhanced`.

Ecranul `vi` conţine un buffer – zona în care apare textul în curs de editare – şi o linie de comandă, în partea de jos a ecranului. Pe această linie sunt afişate informaţii de stare şi pot fi introduse comenzi. În zona bufferului, liniile care sunt goale (spre exemplu, atunci când textul are mai puţine linii decât ecranul) vor conţine caracterul *tilda*, „~”.

Editorul `vi` are trei moduri de lucru: modul de inserare, utilizat pentru scrierea textului, modul de comandă, folosit pentru a „naviga” în cadrul textului, pentru a salva fişierul în curs de editare, pentru a marca blocuri de text etc., şi modul numit *ex*, care se utilizează pentru funcţii adiţionale, cum ar fi căutarea unui şir de caractere în cadrul textului etc.

`vi` porneşte în mod comandă. Pentru a intra în modul de inserare se foloseşte tasta „`i`” (*Insert*). Pentru întoarcere la modul de comandă se apasă tasta `Esc` (*Cancel*). Tastele mai importante de navigare în cadrul textului (valabile în modul de comandă) sunt:

- `h` un caracter la stânga
- `j` o linie în jos
- `k` o linie în sus
- `l` un caracter la dreapta
- `w` un cuvânt înainte
- `b` un cuvânt înapoi
- `$` sfârşitul liniei
- `0` începutul liniei
- `G` începutul bufferului

Alte comenzi importante:

- `x` şterge un caracter
- `d` şterge un obiect. Prin obiect se înţelege un bloc de text, desemnat cu ajutorul comenzilor de navigare. Spre exemplu, comanda `dw` înseamnă „şterge următorul cuvânt".
- `dd` şterge linia
- `D` şterge textul dintre cursor şi sfârşitul liniei
- `u` reface ultima modificare adusă textului

Comenzile pentru ieşierea din `vi`:

- `:wq` salvează fişierul înainte de ieşire
- `:q` iese din `vi` doar dacă nu s-au făcut modificări şi nu au fost salvate
- `:q!` iese din program chiar dacă modificările nu au fost salvate

Comenzile pentru salvarea fişierului sunt:

- `:w` salvează fişierul pe disc
- `:w` *`fisier`* salvează bufferul în fişierul specificat

Mai multe informaţii despre `vi` pot fi obţinute din pagina de manual `man vi`.

Capitolul 2

COMPILATOARE

Eu am o regulă simplă în viaţă:
dacă nu înţeleg ceva, înseamnă că este rău.
(Linus Torvalds)

Compilarea programelor C este formată din trei faze independente, realizate de obicei de trei programe distincte:

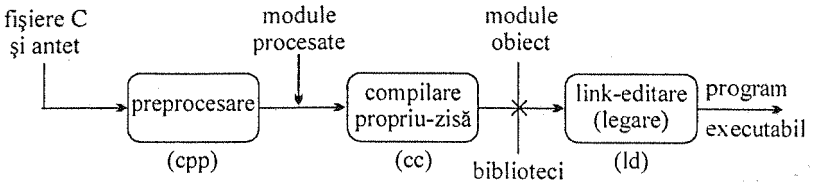


Figura 2.1. Fazele compilării

Utilizând anumite opţiuni (vezi paragraful următor), compilatorul poate fi determinat să parcurgă doar anumite faze.

Preprocesarea

Preprocesorul primeşte un text şi generează tot un text. El analizează macrourile (care încep cu caracterul „#", vezi partea I, capitolul 8 pentru detalii) şi le expandează. Astfel, rezultatul preprocesării este tot un program-sursă C/C++, dar care nu mai include directive preprocesor, ci rezultatul acestora (de exemplu, o directivă `#include` va avea ca efect includerea efectivă a codului C/C++ conţinut în fişierul-antet corespunzător). Faza de preprocesare poate fi apelată prin:

```
gcc -E main.c > main.i
```

Compilarea

Rezultatul compilării este un *fişier-obiect* care reprezintă un amestec de cod maşină şi alte informaţii necesare în faza de editare de legături (link-editare).

Fişierul-obiect conţine în primul rând o listă a simbolurilor nesatisfăcute (adică declarate dar nedefinite), precum şi locurile în care acestea sunt utilizate. Aceste simboluri sunt externe (definite în alte module). De asemenea, fişierul-obiect conţine şi o listă a simbolurilor exportate de către modul.

Fişierul-obiect mai conţine şi *informaţii de relocare*, acestea fiind necesare pentru că toate salturile se fac la nişte adrese care se vor modifica în faza de editare de legături.

Nu în cele din urmă, obiectele pot conţine şi informaţii necesare pentru depanarea programului rezultat (vezi subcapitolul 3.1). Faza de compilare poate fi apelată cu:

```
gcc -c main.c
```

Rezultatul va fi un fişier-obiect cu numele `main.o`.

Editarea de legături

Legarea modulelor constă în completarea referinţelor obiectelor globale, relocarea codului şi adunarea tuturor modulelor într-un singur fişier executabil. Editarea de legături se face cu ajutorul programului `ld`, dar care nu trebuie apelat separat (lista fişierelor-obiect este foarte mare, fiindcă poate include numeroase biblioteci).

Bibliotecile C

O *bibliotecă* este o colecţie de module-obiect deja compilate, strânse la un loc într-un *fişier-arhivă*. În general, declaraţiile globale (variabile, constante, funcţii etc.) se găsesc într-un fişier-antet furnizat împreună cu fişierul-arhivă. Crearea bibliotecilor se face cu ajutorul programului `ar` (pentru detalii vezi subcapitolul 2.3).

Există posibilitatea de a crea biblioteci speciale, denumite *biblioteci partajate* (*shared libraries*). Există un număr mare de funcţii, în special în bibliotecile standard, foarte des utilizate în majoritatea programelor. Funcţiile trebuie încărcate în memorie o singură dată de către sistemul de operare, pentru a nu lega codul lor la toate executabilele, ceea ce ar duce la mărirea dimensiunii acestora. Ca rezultate pozitive, putem enumera: reducerea drastică a dimensiunii executabilelor, micşorarea timpului de încărcare a acestora şi reducerea consumului de memorie.

În Linux, bibliotecile partajate sunt stocate în directoarele `/lib` şi `/usr/lib`.

2.1. Compilarea programelor C: *GCC*

Cel mai răspândit compilator din lumea UNIX este GCC, implementare *open source* realizată de către Free Software Foundation. Acesta este un compilator extensibil, existând extensii pentru C++, Objective-C, Pascal, Fortran etc. Executabilele generate de către acest compilator nu trebuie neapărat să fie *open source*, chiar dacă includ librăriile standard C sau C++.

Prima versiune a compilatorului GCC a fost scrisă de către Richard Stallman, iniţiatorul proiectului GNU, prin 1986.

Sintaxa generală de apelare a compilatorului este:

```
gcc [ optiuni nume_fisier ]
```

Tipul fişierelor de intrare este determinat după sufixul acestora, care sunt prelucrate după cum urmează:

- `.c` – sursă C: preprocesare, compilare, asamblare;
- `.C` – sursă C++: preprocesare, compilare, asamblare;
- `.cc` – sursă C++: preprocesare, compilare, asamblare;
- `.cxx` – sursă C++: preprocesare, compilare, asamblare;
- `.m` – sursă Objective-C: preprocesare, compilare, asamblare;
- `.i` – sursă C preprocesată: compilare, asamblare;
- `.ii` – sursă C++ preprocesată: compilare, asamblare;
- `.s` – sursă în limbaj de asamblare: asamblare;
- `.S` – sursă în limbaj de asamblare: preprocesare, asamblare.

Fişierele având alte sufixe, cum ar fi `.o` (fişier-obiect) sau `.a` (fişier-arhivă), sunt trimise către link-editor.

Pentru a se evita o parte din etapele prelucrării pot fi utilizate următoarele opţiuni:

- `-c` compilează şi asamblează fişierele-sursă, dar nu le link-editează;
- `-S` compilează fişierele-sursă, fără însă a le asambla;
- `-E` nu lansează compilarea, ci doar preprocesarea.

Implicit, `gcc` generează un fişier executabil având denumirea `a.aout`; fişierele-obiect corespunzătoare intrării *`sursa`*`.sufix` sunt denumite *`sursa`*`.o`, iar fişierele asamblate *`sursa`*`.s`.

Denumirea ieşirii poate fi schimbată cu ajutorul opţiunii `-o` *`fisier`*.

Spre exemplu, pentru compilarea unui program C:

```
gcc -o cap7_1 cap7_1.c
```

Alte opţiuni des utilizate:

Opţiuni la compilare:

- `-I`*`director`* adaugă *director* la lista directoarelor în care sunt căutate fişierele `.h`;
- `-L`*`director`* adaugă *director* la lista directoarelor în care sunt căutate bibliotecile;
- `-g` include informaţii de depanare în fişierul-obiect generat. Această opţiune este necesară dacă se intenţionează depanarea programului cu GDB, de exemplu;

- `-O, -O1, -O2, -O3` activează diverse niveluri de optimizare a codului. `-O3` este nivelul cel mai avansat de optimizare;
- `-O0` dezactivează optimizarea codului.

Opţiuni la editarea de legături:

- `-lbiblioteca` utilizează biblioteca specificată în cadrul etapei de editare de legături.

Opţiuni la preprocesare:

- `-Dmacro` defineşte macroul specificat ca fiind `1`;
- `-Dmacro=valoare` defineşte macroul cu valoarea specificată;
- `-Umacro` anulează macroul specificat.

Pentru exemplificare, compilarea unui program care utilizează biblioteca `ncurses`:

```
gcc -o test test.c -I/usr/include/ncurses -lncurses
```

2.2. Compilarea programelor C++: *G++*

Compilatorul G++ se utilizează pentru compilarea programelor C++. Sintaxa de utilizare este identică cu cea a GCC.

Opţiunile mai des utilizate sunt:

- `-fall-virtual` tratează toate funcţiile membre ca fiind virtuale;
- `-fthis-is-variable` permite utilizarea pointer-ului `this` ca o variabilă obişnuită;
- `-fexternal-templates` produce cod obiect mai mic pentru declaraţiile de template-uri, generând doar o singură copie a fiecărei funcţii template acolo unde aceasta este definită. Pentru a putea utiliza această opţiune, trebuie marcate toate fişierele-sursă care utilizează template-uri cu directiva `#pragma implementation` (acolo unde sunt definite efectiv template-urile), respectiv `#pragma interface` (acolo unde sunt declarate template-urile). Atunci când este utilizată această opţiune, toate instanţierile de template-uri sunt considerate `external`. De aceea, toate aceste instanţieri trebuie să fie realizate în cadrul fişierului marcat cu `implementation`, de exemplu prin utilizarea unor declaraţii `typedef` care să facă referire la fiecare instanţiere;

- `-falt-external-templates` are un comportament similar cu cel al opţiunii precedente, cu excepţia faptului că este generată o singură copie a fiecărei funcţii template acolo unde aceasta este definită pentru prima oară.

2.3. Crearea de biblioteci: *ar*

Utilitarul `ar` se foloseşte pentru a crea, modifica şi extrage fişiere dintr-o arhivă, în speţă bibliotecă de tip static.

Pentru a crea o bibliotecă statică, trebuie mai întâi generate fişierele-obiect, apoi creată arhiva:

```
ar -r libtest.a functii.o test.o
ranlib libtest.a
```

Prima comandă va crea arhiva `libtest.a`, conţinând fişierele-obiect `functii.o` şi `test.o`. Cea de-a doua comandă generează un index pentru arhiva nou creată.

Sintaxa programului `ar` este următoarea:

```
ar comanda arhiva [membri...]
```

unde *comandă* poate fi:

- `d` şterge module din arhivă;
- `p` afişează membrii specificaţi (sau toţi membrii dacă nu este specificat nici unul) din arhivă;
- `r` inserează un nou membru în arhivă;
- `t` afişează conţinutul unei arhive;
- `x` extrage un membru din arhivă.

Pentru a crea biblioteci partajate, se va folosi compilatorul `gcc` (sau `g++` pentru programe C++):

```
gcc -shared -o libtest.so functii.o test.o
```

Bibliotecile partajate trebuie instalate în genere în directorul `/usr/lib`, apoi trebuie executată comanda `ldconfig` pentru a actualiza baza de date de biblioteci (*cache*).

Capitolul 3

INSTRUMENTE DE PROGRAMARE

Ideea că există o singură modalitate corectă de a rezolva orice problemă pentru oricare utilizator este fundamental greşită.
(Bjarne Stroustrup)

3.1. Depanarea programelor: *GDB*

Un program de depanare, cum ar fi GDB, se utilizează pentru a putea monitoriza ce se întâmplă într-un program în timpul execuţiei sale sau în momentul în care şi-a încheiat execuţia în mod forţat (*crashed*). Autorul original al programului GDB este Richard Stallman, care a început dezvoltarea acestuia în anul 1988.

Pentru folosirea acestui program, trebuie instalat pachetul `gdb`.

GDB poate fi utilizat atât pentru depanarea programelor scrise în C cât şi a celor scrise în C++. Acesta poate opri din execuţie un program în momentul întâlnirii unei condiţii specificate de programator, examina conţinutul variabilelor la momentul respectiv, şi chiar modifica valoarea acestora şi continua execuţia programului, pentru a putea descoperi mai uşor erorile din program în anumite condiţii. GDB poate fi folosit şi pentru a depana programe care sunt executate pe maşini aflate la distanţă.

Pentru ca un program să poată fi depanat, trebuie să fie compilat cu opţiunea `-g`, care activează includerea informaţiilor de depanare în fişierul-obiect generat.

Modalitatea cea mai simplă de a lansa în execuţie GDB este de a-i furniza ca parametru numele fişierului executabil al programului pe care dorim să îl executăm:

```
gdb program
```

sau:

```
gdb numar_proces
```

pentru a depana procesul specificat. De asemenea, pentru a putea trimite parametri programului apelat, se foloseşte sintaxa:


```
gdb --args program argumente
```

Pentru a executa programul într-un alt terminal decât cel curent, pe care va rula GDB, se utilizează opţiunea `-tty dispozitiv`. Aceasta este utilă dacă, spre exemplu, programul utilizează biblioteca `ncurses` şi face afişarea pe tot ecranul, caz în care ar fi imposibilă execuţia acestuia pe acelaşi terminal cu GDB. Sub Linux, este uşor de comutat între terminale. Ca exemplu, GDB poate fi lansat în execuţie de pe consola virtuală 1 (`tty1`), iar programul să aibă ieşirea şi intrarea pe consola virtuală 7 (`tty7`). Comutarea se face cu ajutorul combinaţiei de taste `Alt+F1`, respectiv `Alt+F7`. De asemenea, această facilitate este utilă pentru a executa programul sub o altă fereastră `X Window`. Pentru a stabili terminalul pe care va rula programul după intrarea în GDB, se utilizează comanda internă `tty terminal` (vezi *infra*).

GDB are o interfaţă de tip *shell*, care permite execuţia de comenzi interne. Pot fi accesate cu ajutorul tastelor săgeţi *sus*, respectiv *jos* comenzile GDB apelate în trecut. Promptul afişat pentru a arăta că se aşteaptă comenzi este `(gdb)`. Ieşirea din GDB se face prin comanda `quit`.

Lansarea în execuţie a programului se face prin comanda `run`. În cadrul GDB, un program poate fi oprit datorită unui semnal, unui punct de oprire (*breakpoint*) sau la solicitarea persoanei care îl depanează. Starea programului poate fi consultată în orice moment cu ajutorul comenzii `info program`.

Puncte de oprire

Un *punct de oprire* (*breakpoint*) opreşte programul atunci când ajunge la un anumit punct din codul-sursă. Acesta se defineşte cu ajutorul comenzii `break`. Astfel, `break functie` va seta un punct de oprire la intrarea în funcţia specificată. Comanda `break numar_linie` stabileşte un punct de oprire în cadrul fişierului-sursă curent. În mod similar, `break fisier:numar_linie` respectiv `break fisier:functie` stabilesc puncte de oprire în linia, respectiv funcţia din fişierul specificat. `break` fără argumente setează un punct de oprire la următoarea instrucţiune care se va executa. Comanda `info breakpoints` afişează punctele de oprire definite.

Un *punct de verificare* (*watchpoint*) opreşte programul atunci când se modifică valoarea unei expresii, fără a şti cu precizie locul unde se întâmplă acest eveniment. Asemenea puncte de verificare se stabilesc prin comanda `watch expresie`. Atunci când expresia este întâlnită în program şi valoarea ei se modifică, programul va fi oprit. Comanda `info watchpoints` afişează punctele de verificare definite.

Un *punct de interceptare* (*catchpoint*) opreşte programul atunci când intervin evenimente speciale, cum ar fi o excepţie C++ sau încărcarea unei biblioteci.

Comanda `catch throw` va opri programul atunci când se tratează o excepţie C++, `catch catch` când se întâlneşte o interceptare a unei excepţii C++.

Pentru a şterge punctele de oprire, de verificare sau de interceptare, se foloseşte comanda `clear`, cu sintaxa `clear [fisier:]functie`, `clear [fisier:]functie` sau `clear [interval]` care şterge intervalul de puncte de oprire specificate, sau toate punctele dacă intervalul nu este specificat.

GDB permite de asemenea detectarea semnalelor UNIX apărute în timpul execuţiei programului şi modificarea comportamentului său la întâlnirea unui asemenea semnal. Astfel, comanda `handle semnal mod_tratare` stabileşte modul de tratare a semnalului specificat. Acesta poate fi specificat atât prin valoarea sa numerică, cât şi prin numele său. Poate lua şi valoarea `all`, care desemnează toate tipurile de semnale. Parametrul `mod_tratare` poate lua una dintre următoarele valori:

- `nostop` programul nu se va opri la apariţia semnalului;
- `stop` programul se va opri la apariţia semnalului;
- `print` GDB va afişa un mesaj la apariţia semnalului;
- `noprint` nu afişează nici un mesaj;
- `pass` permite ca programul să detecteze semnalul. Este posibil ca execuţia programului să se încheie dacă acesta nu a prevăzut tratarea respectivului semnal;
- `ignore` nu permite programului să detecteze semnalul.

Continuarea execuţiei programului

Continuarea execuţiei programului atunci când acesta s-a oprit se face cu ajutorul comenzii `continue`. Comanda `step` continuă execuţia programului până la următoarea linie de cod. Comanda `next` este similară cu `step`, cu excepţia faptului că apelurile de funcţii sunt executate fără oprire (cu alte cuvinte, fără a intra în funcţia respectivă).

Inspectarea variabilelor

În orice moment, pot fi inspectate valorile variabilelor din cadrul programului. Aceasta se face cu ajutorul comenzii `print expresie`. Comanda poate primi ca parametru şi o expresie (în limbajul C), nu neapărat o variabilă. Este permisă şi utilizarea operatorului de rezoluţie, : : din C++. Variabilele statice dintr-un anumit fişier sau funcţie pot fi inspectate folosind formatul `fisier::variabila`, respectiv `functie::variabila`. Spre exemplu, pentru a vizualiza valoarea variabilei globale `count`, definite în fişierul `xobject.cc`:

```
print 'xobject.cc'::count
```


Vom reda în continuare un exemplu de depanare a programului `atoi`, prezentat în paragraful 4.2:

```
$ gcc -g -o atoi atoi.c
$ gdb atoi
GNU gdb Red Hat Linux (5.1.90CVS-5)
Copyright 2002 Free Software Foundation, Inc.
(gdb) break 14
Breakpoint 1 at 0x8048499: file atoi.c, line 14
(gdb) run
Starting program: /home/dragos/atoi

Breaking 1, atoi (s=0x8048598 "1952") at atoi.c:14
14      for(n = 0; s[i] >= '0' && s[i] <= '9'; i++)
(gdb) print i
$1 = 0
(gdb) continue
Continuing.
x = 1952

Program exited normally.
(gdb) quit
```

Mai multe informaţii despre GDB pot fi obţinute apelând comanda `info gdb`.

3.2. Utilitarul *make*

Utilitarul `make` se foloseşte în principal pentru a întreţine un set de fişiere cu interdependenţe, care fac parte dintr-un proiect software. `Make` oferă un sistem de codificare a relaţiilor de legătură dintre fişiere precum şi a comenzilor care trebuie utilizate pentru prelucrarea acestor fişiere atunci când sunt modificate. Acest utilitar serveşte la automatizarea compilării programelor complexe. Majoritatea programelor distribuite în format-sursă din Linux utilizează acest utilitar. Prima implementare a GNU `make` a fost realizată de Richard Stallman şi Roland McGrath, dezvoltarea fiind continuată de către Paul D. Smith.

În vederea utilizării programului `make`, trebuie instalat pachetul cu acelaşi nume.

Sistemul constă dintr-un *fişier makefile*. Acesta conţine mai multe linii, care pot fi de patru tipuri: scop (*target*), comenzi shell, macrouri şi directive. Comentariile încep cu caracterul „#".

La apelarea comenzii `make`, aceasta caută în directorul curent mai întâi fişierul `makefile`, iar apoi fişierul `Makefile`.

Linii de scop

Acestea specifică ceea ce poate fi construit (*e.g.* compilat). Liniile de scop constau dintr-o listă de fişiere scop, urmate de caracterul „:", apoi de lista de dependinţe. Această listă poate fi şi vidă. De exemplu:

```
doc++: comment.cc main.cc readfiles.cc
```

Lista de dependinţe conţine scopuri care trebuie să fie mai vechi decât scopul curent după execuţia comenzii make. Scopul trebuie să fie mai recent decât scopurile de care depinde. Dacă vreuna dintre dependinţe este mai recentă decât scopul curent sau dacă dependinţa nu există, aceasta trebuie construită mai întâi. Dacă lista de dependinţe este vidă, scopul este întotdeauna construit.

În cadrul listei de dependinţe pot fi utilizate şi caractere wildcard, similare cu cele utilizate în interpretorul de comenzi (*shell*) Bash: „*", „?" şi „[]":

```
doc++: *.cc
```

Fişierele din lista de dependinţe pot fi şi de tip obiect, de tip header etc., însă nu este obligatoriu ca acestea să intre în componenţa scopului (*e.g.* executabilului).

Reguli

Make oferă posibilitatea utilizării unor scopuri generice, cunoscute sub numele de *reguli de sufix*, *reguli de inferenţă* sau doar *reguli*. Regulile sunt modalităţi de a specifica doar o singură dată modul de construire al unui scop. Spre exemplu, regula de mai jos specifică modul de transformare a unui fişier-sursă `.cc` într-un fişier-obiect `.o`:

```
# genereaza fisierele obiect .o prin compilarea
# tuturor fisierele-sursa cu extensia .cc
.cc.o:
    g++ -c $< -I. -I/usr/local/include
# genereaza fisierul executabil doc++
# prin link-editarea fisierelor-obiect componente
doc++: comment.o main.o readfiles.o
    g++ comment.o main.o readfiles.o
```

construcţie care este echivalentă cu:

```
# compileaza comment.cc
comment.o: comment.cc
    g++ -c comment.cc -I. -I/usr/local/include
# compileaza main.cc
main.o:     main.cc
    g++ -c main.cc -I. -I/usr/local/include
# compileaza readfiles.cc
readfiles.o:  readfiles.cc
    g++ -c readfiles.cc -I. -I/usr/local/include
# genereaza fisierul executabil doc++
# prin link-editarea fisierelor obiect componente
main:  comment.o main.o readfiles.o
    g++ comment.o main.o readfiles.o
```


În exemplul de mai sus a fost utilizat un macrou special, `$<`, care substituie fişierul-sursă curent.

Tipul fişierului este determinat după extensia sa (şirul de caractere de după punct „."). Dacă există vreun fişier de tipul respectiv, regula de sufix va fi aplicată.

Linii de comenzi shell

Liniile de comenzi shell, cunoscute şi sub numele de comenzi, definesc acţiunile care sunt executate pentru a construi un scop. Textul care urmează după caracterul „;" în cadrul unei linii de scop este considerat o comandă. De asemenea, toate liniile care urmează după un scop şi care încep cu caracterul *tab* sunt considerate comenzi care vor fi executate în vederea construirii scopului.

Liniile lungi de comenzi pot fi continuate pe linia următoare adăugând la sfârşit caracterul „\".

În mod normal, comenzile sunt afişate la ieşirea standard. Acest lucru poate fi prevenit prin prefixarea comenzii cu caracterul *at*, „@".

Macrouri

Macrourile au următoarele roluri:

- scutesc munca programatorului care scrie fişierul makefile;
- uşurează întreţinerea fişierului permiţând specificarea informaţiilor într-un singur loc. Atunci când valoarea unui macrou se modifică, aceasta se propagă în mod automat în tot fişierul. Spre exemplu, lista de fişiere-obiect care vor fi construite poate fi specificată prin intermediul unui macrou;
- oferă un mod de a introduce variabile într-un makefile, cunoscând informaţiile care vor fi modificate. De exemplu, modul de compilare a unui program cu sau fără informaţii de depanare poate fi determinat printr-un macrou. Modificând valoarea acestuia, în linia de comandă sau printr-o variabilă de mediu, modul de compilare poate fi stabilit fără a modifica fişierul makefile;
- uşurează posibilitatea de citire a fişierului makefile, reducând numărul de caractere conţinute.

Macrourile pot fi definite prin definiţii interne fişierului makefile, prin variabile de mediu, sau în linia de comandă. În cadrul fişierului acestea pot fi definite astfel:

```
nume = valoare
```

Numele macrourilor sunt caractere mari prin convenţie. *valoare* poate fi şi o listă de valori. De asemenea, definiţiile pot include alte macrouri:

```
INCLUDES=/usr/local/include /usr/include/Xterminal
OBJECTS=test.o xterminal.o
OBJS=$(OBJECTS) main.o
```

După cum s-a putut observa şi din exemplul de mai sus, macrourile pot fi referite prin intermediul construcţiilor:

```
$(macro)
${macro}
```

Macrouri predefinite

Utilitarul make oferă o serie de macrouri speciale, predefinite, pentru a facilita utilizarea regulilor generice. Aceste macrouri pot fi referite fără a mai utiliza parantezele. De asemenea, acestora nu le poate fi atribuită nici o valoare, aceasta depinzând de starea în care se află make în momentul în care macroul este evaluat. Macrourile predefinite sunt următoarele:

Macrou	Descrierea macroului
$@	Numele scopului curent
$%	Valoarea membrului curent al arhivei, doar în cazul în care scopul curent este o bibliotecă
$?	Lista de dependinţe care sunt învechite pentru scopul curent
$<	Fişierul-sursă curent, altfel spus fişierul care se prelucrează în momentul întâlnirii acestui macrou. Fişierul-sursă curent reprezintă fişierul care este învechit în comparaţie cu scopul curent
$*	Rădăcina scopului curent. Spre exemplu, dacă scopul se numeşte `readfiles.o`, rădăcina are sufixul `.o` şters, deci este `readfiles`
$$@	Este similar cu `$@`, însă se utilizează pentru executabilele alcătuite dintr-un singur fişier-sursă

Mai există câteva macrouri predefinite, dar care pot fi modificate de către programator, dintre care mai importante sunt:

Macrou	Descrierea macroului
VPATH	Lista de directoare în care make va căuta fişierele dependinţă. Elementele listei se despart cu ajutorul caracterului „:”
SHELL	Interpretorul de comenzi utilizat pentru a executa liniile de comandă

Directive

Fiecare implementare a utilitarului make (sub Linux se utilizează GNU make, `gmake`) conţine propriile directive. Directivele „#” (comentariu) şi `include` sunt prezente totuşi în toate implementările. Astfel, directiva `include` se utilizează pentru a include într-un makefile definiţiile făcute într-un alt makefile, pentru a reduce redundanţa acestora.

Argumente în linia de comandă

Argumentele uzuale sunt:

- `-f` *`fisier`* stabileşte numele fişierului makefile care va fi procesat;
- `-p` afişează toate definiţiile de macrouri şi regulile;
- `-n` nu execută comenzi, doar le afişează la ieşirea standard;
- `-s` activează modul de lucru silenţios, în sensul că la ieşire comenzile nu vor fi afişate.

Mai multe informaţii privind make pot fi obţinute apelând `info make`.

Iată un exemplu de fişier makefile care are drept scop „compilarea” unui fişier LaTeX în vederea obţinerii documentului Postscript final:

```
# fisier folosit pentru compilarea cartii
# versiunea 0.9 - Sabin Corneliu Buraga <busaco@infoiasi.ro>
# ultima actualizare: 17 iulie 2001

# fisierul principal al cartii
SRC = main

# operatiunile complete (make)
all: tex idx ps

# doar compilare (make tex)
tex: $(SRC).tex
    latex $(SRC).tex

# generare index (make idx)
idx:
    makeindex $(SRC).idx
    latex $(SRC).tex

# generare PostScript (make ps)
ps:
    dvips $(SRC).dvi -o $(SRC).ps

# generare PDF (make pdf)
pdf:
    ps2pdf $(SRC).ps

# vizualizare PostScript cu gv (make gv)
gv:
```

```
    gv $(SRC).ps &

# arhiveaza fisierele (make arh)
arh:
    tar -zcf retele-$(SRC).tgz *.tex Makefile *.sty *.cls eps/*

# dezarhiveaza fisierele (make unarh)
unarh:
    tar -zxf retele-$(SRC).tgz

# sterge fisierele inutile (make clean)
clean:
    rm *.aux *.dvi *.i?? *.l?? *.toc *~
```

3.3. *autoconf, automake* şi *libtool*

Programele `autoconf`, `automake` şi `libtool` sunt utile pentru a simplifica compilarea aplicaţiilor complexe pe diferite platforme şi pentru a creşte gradul de portabilitate a acestora.

`Autoconf` uşurează portarea programelor pe diferite platforme prin determinarea caracteristicilor sistemului pe care se face compilarea înainte ca aceasta să înceapă (versiunile bibliotecilor instalate, existenţa anumitor biblioteci etc.). Codul-sursă se poate adapta astfel mai uşor la diferenţele existente între platforme. Prima versiune `autoconf` a fost realizată în anul 1992 de către David MacKenzie.

Pentru a putea folosi aceste programe, trebuie instalate pachetele cu aceleaşi nume.

`Automake` este un instrument care generează în mod automat fişiere makefile, simplificând mult munca de realizare a acestora. El oferă de asemenea diverse funcţii utile precum determinarea automată a dependenţelor dintre fişiere. Dezvoltarea programului `automake` a fost începută în anul 1994 de către David MacKenzie şi Tom Tromey.

`Libtool` este o interfaţă pentru compilator şi link-editor care permite generarea uşoară a bibliotecilor, fie ele statice sau dinamice, independent de platformă. Utilitarul `libtool` a fost creat de către Gordon Matzigkeit în anul 1996.

Aceste trei instrumente sunt programe distincte, însă între ele există o legătură strânsă (vezi *infra*).

Vom numi în continuare un arbore de cod-sursă a unui program, împreună cu alte fişiere cum ar fi documentaţii, fişiere de configurare etc., *pachet*. Un pachet construit cu `autoconf` conţine un script denumit `configure`. Pentru a construi (compila) programul, trebuie executat mai întâi acest script pentru a determina particularităţile sistemului şi a pregăti codul-sursă în acest sens. Pentru compilare se va apela comanda `make`.

Opţiunile cele mai des utilizate la apelarea scriptului `configure` sunt:

- `--prefix=`*director* stabileşte directorul în care vor fi instalate fişierele independente de arhitectură ale pachetului (*e.g.* documentaţii);
- `--exec-prefix=`*director* stabileşte directorul în care vor fi instalate fişierele dependente de arhitectură, cum ar fi executabile sau biblioteci. Dacă nu este folosită, va fi utilizată valoarea furnizată opţiunii `--prefix`;
- `--bindir=`*director* specifică destinaţia în care vor fi instalate fişierele executabile care vor fi apelate de către utilizatori;
- `--libexecdir=`*director* specifică directorul în care vor fi instalate bibliotecile utilizate de către programele generate;
- `--libdir=`*director* atunci când pachetul furnizează biblioteci, această opţiune stabileşte directorul în care acestea vor fi instalate;
- `--disable-`*opţiune* dezactivează includerea opţiunii (*feature*) specificate. Aceste opţiuni pot activa anumite porţiuni de cod sau implica compilarea condiţionată a unor chestiuni experimentale, de exemplu `--disable-debug` va dezactiva generarea informaţiilor pentru depanare;
- `--enable-`*opţiune* este similară cu opţiunea precedentă, dar activează opţiunea specificată. Există posibilitatea ca această opţiune să primească un argument, de exemplu `--enable-buffers=64`;
- `--with-`*pachet* este similară cu opţiunea `--enable-`*optiune*, dar se referă în general la includerea unei biblioteci sau a unui alt pachet;
- `--without-`*pachet* opusul opţiunii `--with-`*pachet*.

La execuţia scriptului `configure`, sunt generate o serie de fişiere şi eventual directoare. Fişierele generate sunt:

- `config.cache` memorează variabilele dependente de sistem generate, pentru a evita determinarea lor de mai multe ori;
- `config.log` conţine jurnalul testelor şi rezultatelor acestora, incluzând comenzile shell apelate şi mesajele generate;
- `config.status` este un script în care fişierele sunt generate conform configurărilor curente;
- `config.h` fişier-antet (C/C++) care conţine directive preprocesor care descriu configurările curente. Acestea sunt necesare pentru programele din cadrul pachetului pentru a determina respectivele configurări. Iată un fragment dintr-un asemenea fişier-antet:

```
/* Define if you have the <unistd.h> header file. */
#define HAVE_UNISTD_H

/* Name of the package */
#define PACKAGE "doc++"

/* Version number of the package */
```

```
#define VERSION "3.4.9"

/* Define if the GNU gettext() function is already present or
preinstalled */
#define HAVE_GETTEXT 1
```

- `Makefile` fişierul makefile care va fi utilizat de comanda `make` pentru a construi fişierele necesare. În general este generat pornind de la fişierul `Makefile.in`.

În cadrul fişierului makefile generat, sunt definite câteva scopuri în vederea manipulării pachetului:

- `make all` construieşte toate fişierele necesare;
- `make install` instalează pachetul;
- `make clean` şterge toate fişierele generate.

Pentru a înţelege mai uşor structura şi modul de funcţionare a perechii de programe autoconf-automake, vom considera un proiect ipotetic denumit `aa`.

Scriptul `configure` este generat în mod automat cu ajutorul comenzii `autoconf`, pornind de la un fişier denumit `configure.in`. Acesta conţine macrouri şi cod shell.

Iată în continuare conţinutul fişierului `configure.in` pentru proiectul în discuţie:

```
dnl Comentariile incep cu dnl
dnl Acest fisier trebuie procesat cu ajutorul comenzii autoconf
dnl pentru a produce scriptul de configurare
AC_INIT(main.c)
AM_INIT_AUTOMAKE(aa, 1.0)
AC_PROG_CC
AC_OUTPUT(Makefile)
```

Macroul `AC_INIT` realizează o serie de iniţializări şi stabileşte ca fişier principal al proiectului `main.c`. Acesta este utilizat pentru a determina dacă fişierele-sursă se găsesc în directoarele corespunzătoare. Apelul acestui macrou este obligatoriu.

Macroul `AC_INIT_AUTOMAKE` iniţializează `automake`, specificând numele proiectului şi versiunea curentă. Şi apelul acestui macrou este obligatoriu dacă se utilizează `automake`.

Macroul `AC_PROG_CC` verifică instalarea corectă a compilatorului de C. Macroul `AC_OUTPUT` stabileşte care este fişierul care va fi generat. Fişierul makefile se generează pornind de la un fişier makefile „şablon", denumit `Makefile.in`.

Fişierul `Makefile.in` este generat în mod automat de către utilitarul `automake` pornind de la fişierul denumit `Makefile.am`. Mai jos prezentăm conţinutul acestui fisier pentru proiectul nostru:


```
bin_PROGRAMS = aa
aa_SOURCES = main.c functii.c functii.h
aa_LDADD = -lncurses
```

Prima directivă, `bin_PROGRAMS`, specifică lista de programe executabile care vor fi generate. A doua directivă, având formatul *`binar`*`_SOURCES`, specifică fişierele-sursă necesare pentru a construi executabilul în cauză, inclusiv cele antet. În final, directiva `aa_LDADD` desemnează bibliotecile care trebuie link-editate împreună cu executabilul final.

După crearea acestor două fişiere, trebuie executată comanda:

```
aclocal; autoconf
```

Comanda `aclocal` se apelează pentru a procesa macrourile care nu sunt cunoscute de `autoconf`, cum este în cazul nostru macroul `AC_INIT_AUTOMAKE`. Această comandă generează scriptul `aclocal.m4`, în limbaj `M4`.

După aceasta, vom apela comanda `automake --add-missing` pentru a crea fişierul `Makefile.in` şi a copia câteva scripturi care vor fi necesare ulterior. Opţiunea menţionată a fost inclusă deoarece în directorul curent al proiectului nostru nu se află o serie de fişiere standardizate prin convenţie:

- `AUTHORS` lista de persoane, însoţite de regulă şi de adresele lor de e-mail, care au contribuit la proiect;
- `ChangeLog` jurnalizează modificările aduse proiectului;
- `README` include informaţii generale despre proiect, inclusiv modul de instalare etc.;
- `NEWS` lista de modificări importante, vizibile utilizatorilor.

Aceste fişiere sunt conforme standardelor GNU, dar nu influenţează în vreun fel funcţionarea programelor `autoconf` sau `automake`.

Este important să notăm aici faptul că, după orice modificare adusă fişierelor de configurare `autoconf` şi `automake`, trebuie apelate comenzile menţionate mai sus pentru a regenera fişierele.

Structurarea apelurilor de comenzi şi macrouri în cadrul fişierului `configure.in` se face în următoarele secţiuni, în ordinea menţionată:

1. Macrourile de bază: `AC_INIT` (care trebuie să fie primul macrou apelat), `AM_INIT_AUTOMAKE`, `AC_CONFIG_HEADER`.
2. Opţiunile din linia de comandă `configure`, cum ar fi `AC_ARG_ENABLE`.
3. Programele care sunt necesare în cadrul procesului de configurare, sau pentru programul care va fi compilat, utilizând macrouri de genul `AC_CHECK_PROG` sau `AC_PATH_TOOL`.
4. Bibliotecile necesare în cadrul testelor din procesul de configurare, sau de programul care va fi compilat, prin macroul `AC_CHECK_LIB`.

5. Fişierele-antet (dacă acestea există).
6. Definiţiile de tipuri sau structuri.
7. Funcţiile utilizate, cu ajutorul macroului `AC_CHECK_FUNCS`.
8. Fişierele generate, cum ar fi fişierele makefile, utilizând macroul `AC_OUTPUT`.

Este bine ca la începutul lui `configure.in` să se apeleze macroul `AC_CANONICAL_HOST`, care determină numele standardizat (*canonical system name*) al sistemului sau *numele configuraţiei*. Aceste nume sunt alcătuite din trei părţi, având formatul *procesor-producător-sistem_de_operare*. De asemenea, pe sistemele în care nucleul diferă de sistemul de operare, cum ar fi GNU/Linux, numele poate avea formatul *procesor-producător-nucleu-sistem_de_operare*. Spre exemplu, numele standardizat pentru un sistem Linux este de genul `i586-pc-linux-gnu`. Alt exemplu este `sparc-sun-solaris2.4`.

Să ne reamintim conţinutul fişierului `Makefile.am` pentru proiectul `aa`:

```
bin_PROGRAMS = aa
aa_SOURCES = main.c functii.c functii.h
aa_LDADD = -lncurses
```

Într-un mod similar cu declararea programelor care vor fi construite (compilate) prin directiva `bin_PROGRAMS`, există posibilitatea realizării şi de alte scopuri, după cum urmează:

- `DATA` defineşte fişierele care fac parte din proiect (cu alte cuvinte vor fi instalate), dar nu necesită vreo prelucrare pentru a fi construite;
- `HEADERS` declară fişierele-antet;
- `SCRIPTS` declară scripturile executabile;
- `MANS` declară fişierele manual ale proiectului. Există două moduri de utilizare a acestui macrou:
  - `man_MANS` atunci când fişierele au deja prefixul care semnifică secţiunea de manual căreia îi aparţin, stabilit, cum ar fi `docify.1`
  - `man1_MANS` atunci când fişierele nu au prefixul stabilit, cum ar fi `equate.man`.

Atunci când proiectul are mai multe subdirectoare, fişierul `Makefile.am` din directorul (sau directoarele) rădăcină trebuie să conţină un macrou de forma:

```
SUBDIRS = doc src
```

Subdirectoarele vor fi procesate în ordinea specificată.

În mod tradiţional, structura unui arbore-sursă pentru un proiect conţine un director denumit `src` pentru fişiere-sursă, un director `doc` pentru documentaţii şi, eventual, un director `test` pentru programe de test.

Vom prezenta în cele ce urmează programul `libtool`.

Pentru a utiliza `libtool`, la începutul fişierului `configure.in` trebuie apelat macroul `AC_PROG_LIBTOOL`. Vor fi disponibile mai multe opţiuni noi, dintre care mai importante sunt:

- `--enable-shared` activează construirea de biblioteci partajate;
- `--disable-shared` este complementul opţiunii precedente;
- `--enable-static` activează construirea de biblioteci statice;
- `--disable-static` este reversul opţiunii precedente.

În cele ce urmează vom presupune că proiectul nostru este o bibliotecă. Fişierul `Makefile.am` va avea următorul conţinut:

```
lib_LTLIBRARIES = libaa.la
libaa_la_SOURCES = functii.cc
```

Se observă similitudinea cu declaraţiile `bin_PROGRAMS`, respectiv `aa_SOURCES`. Proiectul nostru nu conţine decât fişierul-sursă `functii.cc`.

În mod tradiţional, pe lângă directoarele menţionate mai sus, arborele-sursă mai trebuie să conţină şi directorul `include`, care va include fişierele-antet (în general cele care vor fi instalate, adică vizibile altor programatori care vor utiliza biblioteca). Fişierul `Makefile.am` din cadrul acestui director trebuie să conţină o directivă `include_HEADERS`.

După crearea fişierelor `configure.in` şi `Makefile.am`, trebuie copiat conţinutul scriptului `libtool.m4`, integrat în pachetul `libtool` (fişier localizat în general în directorul `/usr/share/libtool/`), în directorul rădăcină al proiectului, sub numele de `acinit.m4`. Pasul următor este execuţia comenzilor:

```
aclocal
libtoolize
```

În urma execuţiei comenzii `libtoolize` poate fi remarcată apariţia unor fişiere noi, `ltconfig` şi `ltmain.sh`.

În cazul bibliotecilor, este importantă gestionarea corectă a versiunilor, deoarece o schimbare de versiune poate aduce şi modificări ale funcţiilor, fişierelor-antet etc. Un program care utilizează biblioteca va avea probleme la execuţie sau este posibil chiar să nu funcţioneze deloc. `libtool` posedă un sistem propriu de control al versiunilor, permiţând chiar existenţa mai multor versiuni de biblioteci instalate.

Formatul versiunii unei biblioteci este:

```
curent:revizie:varsta
```

unde:

- *curent* reprezintă interfaţa curentă exportată de către bibliotecă (conţinutul fişierelor-antet vizibile altor programatori);
- *revizie* reprezintă implementarea curentă a interfeţei exportate (dacă interfaţa rămâne neschimbată dar se modifică implementarea internă a unor funcţii, *curent* va rămâne nemodificată şi va fi incrementat *revizie*);
- *vârstă* semnifică numărul de interfeţe mai vechi suportate de versiunea curentă a bibliotecii (cu alte cuvinte, compatibile cu aceasta). *vârstă* trebuie să fie mai mic sau cel mult egal cu *curent*.

Regulile de modificare a acestor trei câmpuri sunt următoarele:

1. În cazul în care codul-sursă a fost modificat, trebuie incrementat câmpul *revizie*. Vom spune că avem o nouă revizie a interfeţei curente.
2. Dacă interfaţa a fost modificată, *curent* trebuie incrementat, iar *revizie* pus pe zero. Spunem că este vorba de prima revizie a unei noi interfeţe.
3. Dacă noua interfaţă a adăugat noi elemente, dar nu a modificat interfaţa versiunii precedente (deci programele vechi sunt compatibile cu noua versiune), trebuie incrementat câmpul *vârstă*. Spunem aşadar că această versiune este compatibilă cu versiunea precedentă.
4. Dacă în noua interfaţă s-au eliminat componente, programele vechi nu vor mai fi compatibile cu aceasta, şi va trebui ca `varsta` să fie pus pe zero.

Pentru o mai bună înţelegere a utilizării acestor trei programe, vom prezenta în continuare două exemple reale, unul pentru un program executabil şi unul pentru o bibliotecă.

Primul exemplu este programul `SA1`, pentru evidenţa personalului şi a salariilor. Iată structura de directoare a arborelui surselor, în care fişierele care ne interesează au fost îngroşate:

```
.:
acconfig.h config.guess configure doc lista missing
aclocal.m4 config.status configure.in install Makefile.am
mkinstalldirs
build  config.sub devel install-sh Makefile.in src

./build:
coeficienti.data create_global create_luna create_luna_install
impozit.data

./devel:
sal-1.0.spec

./doc:
ChangeLog

./install:
install main.cc Makefile.am
```



```
./src:
bonuri.cc fisafisc1.cc general.h main.h pontaj.cc stat.cc
cas.cc fisafisc2.cc init.cc Makefile.am salarii.cc useradaug.cc
config.h.in fluturi.cc initluna.cc  Makefile.in salcalc.cc
usersterg.cc
conflunian.cc general.cc main.cc personal.cc stamp-h.in
```

Makefile.am:

```
AUTOMAKE_OPTIONS = foreign

SUBDIRS = install src
```

acconfig.h:

```
/* The package name. */
#undef PACKAGE

/* The package version. */
#undef VERSION

/* Define if you have the `crypt' function. */
#undef HAVE_CRYPT

/* Define if you are using `PostgreSQL' as SQL server. */
#undef HAVE_LIBPQ
```

configure.in:

```
dnl configure.in for SA1
dnl Copyright (c) 1999-2001 Dragos Acostachioaie

dnl Declara fisierul principal al proiectului
AC_INIT(src/main.cc)
dnl Declara limbajul in care este realizat proiectul
AC_LANG_CPLUSPLUS
dnl Prefixul implicit de instalare
AC_PREFIX_DEFAULT(/usr)

dnl Versiunea de program
VERSION="1.0"

dnl Initializari automake
AM_INIT_AUTOMAKE(SA1, $VERSION)
dnl Fisierul antet de configurare
AM_CONFIG_HEADER(src/config.h)

dnl Determina numele standardizat al masinii
```

```
AC_CANONICAL_HOST

dnl Verifica existenta programelor
dnl Verifica existenta compilatorului de C
AC_PROG_CC
dnl Verifica existenta compilatorului de C++
AC_PROG_CXX
dnl Verifica existenta programului make
AC_PROG_MAKE_SET

dnl Verifica existenta fisierelor-antet
dnl Verifica existenta fisierelor standard C
AC_HEADER_STDC
dnl Verifica existenta fisierelor-antet getopt.h
dnl si sys/time.h
AC_CHECK_HEADERS(getopt.h sys/time.h)

dnl Verifica existenta bibliotecilor
dnl Verifica existenta bibliotecii crypt
AC_CHECK_LIB(crypt, main, [
    AC_DEFINE(HAVE_CRYPT)
    LIBS="-lcrypt"])

dnl Verifica existenta bibliotecii pq
AC_CHECK_LIB(pq, main, [
    AC_DEFINE(HAVE_LIBPQ)
    LIBS="$LIBS -lpq"],
    AC_MSG_ERROR(This version of SAl requires libpq library))
dnl Stabileste bibliotecile care trebuie link-editate
dnl impreuna cu executabilele generate
LIBS="$LIBS -lXterminal -lncurses -lgpm -lpinguins"
dnl Stabileste optiunile pentru preprocesor
CPPFLAGS="-I/usr/include/Xterminal -I/usr/include/ncurses
-I/usr/include/pgsql -I/usr/include/Pinguins"

dnl Verifica typedef-uri, structuri si caracteristici
dnl compilatorului
AC_HEADER_TIME

dnl Verifica existenta anumitor functii
AC_CHECK_FUNCS(getopt_long gettimeofday)

dnl Stabileste optiunile pentru compilatorul C++
CXXFLAGS="-O2 -w"
if test "$CC" = "gcc"; then
    CXXFLAGS="$CXXFLAGS -funsigned-char -falt-external-templates"
fi

dnl Fisierele care vor fi generate
AC_OUTPUT(Makefile src/Makefile)
```


install/Makefile.am

```
bin_PROGRAMS = config

config_SOURCES = main.cc
```

src/Makefile.am

```
bin_PROGRAMS = sal
sal_SOURCES = bonuri.cc cas.cc conflunian.cc fisafisc1.cc \
              fisafisc2.cc fluturi.cc general.cc init.cc initluna.cc \
              main.cc personal.cc pontaj.cc salarii.cc salcalc.cc \
              stat.cc useradaug.cc usersterg.cc

clean-local:
    rm -f *.lst

distclean-local:
    rm -f *.lst

maintainer-clean-local:
    rm -f *.lst
```

src/config.h.in

```
/* src/config.h.  Generated automatically by configure. */
/* src/config.h.in.  Generated automatically from configure.in by
autoheader. */

/* Define if you have the ANSI C header files. */
#define STDC_HEADERS 1

/*  Define  if  you  can  safely  include  both  <sys/time.h>  and
<time.h>. */
#define TIME_WITH_SYS_TIME 1

/* Define if you have the `crypt' function. */
#define HAVE_CRYPT 1

/* Define if you are using `PostgreSQL' as SQL server. */
#define HAVE_LIBPQ 1

/* Define if you have the getopt_long function. */
/* #undef HAVE_GETOPT_LONG */

/* Define if you have the gettimeofday function. */
/* #undef HAVE_GETTIMEOFDAY */
```

```
/* Define if you have the <getopt.h> header file. */
#define HAVE_GETOPT_H 1

/* Define if you have the <sys/time.h> header file. */
#define HAVE_SYS_TIME_H 1

/* Name of package */
#define PACKAGE "SA1"

/* Version number of package */
#define VERSION "1.0"
```

Al doilea exemplu este biblioteca `Xterminal`, o interfață utilizator în mod terminal, realizată în limbajul C++. Redăm mai jos structura de directoare a arborelui surselor, în care fișierele care ne interesează au fost îngroșate (unele directoare au fost omise pentru acuratețea descrierii):

```
.:
acconfig.h  configure   doc        install-sh Makefile.am   NEWS
aclocal.m4  configure.in example lista     Makefile.in    README
config.guess COPYING.LIB  include ltconfig         missing src
config.sub  CREDITS    INSTALL ltmain.sh   mkinstalldirs

./doc:
ChangeLog manual objects.tree TO-DO

./doc/manual:
[...]

./example:
exapp.cc exdlg.cc exfreeapp.cc exlist.cc extab.cc Makefile.am
exbar.cc exevent.cc exhlist.cc exmyapp.cc exwin.cc Makefile.in

./include:
Makefile.am xevent.h xtdesktop.h xtlist.hxttexteditor.h
Makefile.in xobject.h xtdialog.h  xtmenu.h xttools.h
RegionalInfo.hxtapp.h  xterminal.h xtother.h xtwindow.h
stamp-h.in xtbar.h xtfield.h xtpassive.h
xbclass.h xtbutton.h xthlist.h  xtscreen.h
xconfig.h.in xtcombobox.h xtlib.h xttab.h

./src:
Makefile.am  xobject.cc  xtdesktop.cc xtlist.cc xttab.cc
Makefile.in  xtapp.cc  xtdialog.cc  xtmenu.cc xttexteditor.cc
RegionalInfo.cc xtbar.cc xtfield.cc xtother.cc    xttools.cc
xbclass.cc xtbutton.ccxthlist.cc xtpassive.cc xtwindow.cc
xevent.cc xtcombobox.cc xtlib.cc  xtscreen.cc
```

Makefile.am


```
AUTOMAKE_OPTIONS = foreign

SUBDIRS = include src

clean-local:
    cd example; $(MAKE) clean

distclean-local:
    cd example; $(MAKE) distclean

examples:
    cd example; $(MAKE)
```

acconfig.h

```
/* The package name. */
#undef PACKAGE

/* The package version. */
#undef VERSION

/* Define if you want to catch the signals and translate them into
events. */
#undef SIGNAL_CATCHING

/* Define if you don't want to use that ugly ncurses hacks. */
#undef NO_NCURSES_HACKS

/* Define if you want to generate debugging informations. */
#undef DEBUG

/* Define if you are using `ncurses' as curses library. */
#undef HAVE_NCURSES

/* Define if you are using `slang' as curses library. */
#undef HAVE_SLANG

/* Define if you want to use the Linux console mouse support. */
#undef HAVE_LIBGPM

/* Define if your curses have `ESCDELAY'. */
#undef HAVE_ESCDELAY

/* Define if the use of `TIOCGWINSZ' requires `<sys/ioctl.h>' */
#undef GWINSZ_IN_SYS_IOCTL
```

configure.in

```
dnl configure.in for Xterminal library
dnl Copyright (c) 1997-2001 Dragos Acostachioaie
```

```
dnl Process this file with autoconf to produce a configure script.
dnl Declara fisierul principal al proiectului
AC_INIT(src/xobject.cc)
dnl Declara limbajul in care este realizat proiectul
AC_LANG_CPLUSPLUS
dnl Prefixul implicit de instalare
AC_PREFIX_DEFAULT(/usr)

dnl Stabileste versiunea bibliotecii
VERSION="1.0.6"
CURRENT=16
REVISION=0
AGE=10
AC_SUBST(CURRENT)
AC_SUBST(REVISION)
AC_SUBST(AGE)

dnl Initializari automake
AM_INIT_AUTOMAKE(Xterminal, $VERSION)
dnl Declara fisierul-antet de configurare
AM_CONFIG_HEADER(include/xconfig.h)

dnl Determina numele standardizat al masinii
AC_CANONICAL_HOST

dnl Verifica existenta compilatorului de C
AC_PROG_CC
dnl Verifica existenta compilatorului de C++
AC_PROG_CXX

dnl Initializeaza libtool
dnl Implicit vor fi construite biblioteci dinamice
AM_ENABLE_SHARED
AM_DISABLE_STATIC
dnl Verifica existenta libtool
AM_PROG_LIBTOOL

dnl Verifica existenta fisierelor antet
AC_HEADER_STDC
AC_CHECK_HEADERS(sys/ioctl.h string.h strings.h unistd.h)
AM_HEADER_TIOCGWINSZ_NEEDS_SYS_IOCTL

dnl Verifica typedef-uri, structuri si caracteristici ale
dnl compilatorului
AC_STRUCT_TM

dnl Verificia existenta functiilor
AC_TYPE_SIGNAL
AC_FUNC_VPRINTF
AC_CHECK_FUNCS(strdup strtod)

dnl Stabileste optiunile pentru compilator
CXXFLAGS="-O2 -w"
```



```
dnl Introduce optiunea --disable-signals,
dnl care va dezactiva conversia semnalelor in evenimente
dnl in cadrul bibliotecii
AC_ARG_ENABLE(signals,
    [ --disable-signals Turn off signal translation into events],,
    AC_DEFINE(SIGNAL_CATCHING)
)

dnl Introduce optiunea --disable-ncurses-hacks,
dnl care va dezactiva utilizarea structurilor interne
dnl ale bibliotecii ncurses
AC_ARG_ENABLE(ncurses-hacks,
    [ --disable-ncurses-hacks Don't use ncurses internal structures],
    AC_DEFINE(NO_NCURSES_HACKS)
)

dnl Verifica daca se genereaza informatii
dnl de depanare
AC_ARG_ENABLE(debug,
    [ --enable-debug Turn on debugging],
    AC_DEFINE(DEBUG)
    CXXFLAGS="$CXXFLAGS -g"
)

dnl Verifica daca se utilizeaza slang
dnl ca biblioteca curses
AC_ARG_WITH(slang,
    [ --with-slang Use slang library instead of ncurses],
    withslang=x$withval,
    withslang=xno
)
if test $withslang = xyes; then
    AC_CHECK_LIB(slang, main, [
    AC_DEFINE(HAVE_SLANG)
    LIBS="-lslang"
    CPPFLAGS="$CPPFLAGS -I/usr/include/slang"],
    AC_MSG_WARN(Slang library not found, trying ncurses)
    )
fi

dnl Verifica existenta bibliotecii ncurses
if test "$ac_cv_lib_slang_main" != "yes"; then
AC_CHECK_LIB(ncurses, main, [
    AC_DEFINE(HAVE_NCURSES)
    LIBS="-lncurses"
    CPPFLAGS="$CPPFLAGS -I/usr/include/ncurses"],
    AC_MSG_ERROR(No curses library found)
)
fi

dnl Verifica daca se ofera suport pentru mouse
AC_ARG_WITH(gpm,
    [ --without-gpm Disable GPM mouse support],
    withgpm=x$withval,
```

```
    withgpm=xyes
)
if test $withgpm = xyes; then
    AC_CHECK_LIB(gpm, main, [
    AC_DEFINE(HAVE_LIBGPM)
    LIBS="$LIBS -lgpm"])
fi

dnl Verifica existenta functiei resizeterm
AC_CHECK_FUNCS(resizeterm)

dnl Verifica existenta variabilei globale ESCDELAY
dnl in cadrul bibliotecii curses
AC_MSG_CHECKING(for ESCDELAY support)
if test "$ac_cv_lib_ncurses_main" = "yes"; then
    AC_CACHE_VAL(ac_cv_have_escdelay,
    AC_TRY_LINK([
    #include <curses.h>], [
    {
       ESCDELAY = 0;
    }],
    ac_cv_have_escdelay=yes,
    ac_cv_have_escdelay=no)
    )
fi
if test "$ac_cv_lib_slang_main" = "yes"; then
    ac_cv_have_escdelay=yes
fi
AC_MSG_RESULT($ac_cv_have_escdelay)
if test "$ac_cv_have_escdelay" = "yes"; then
    AC_DEFINE(HAVE_ESCDELAY)
fi

dnl Optiunile pentru compilatorul de C
if test "$CC" = "gcc"; then
    CXXFLAGS="$CXXFLAGS -funsigned-char -fthis-is-variable
       -falt-external-templates"
fi

includedir="$includedir/Xterminal"

dnl Fisierele care vor fi generate
AC_OUTPUT(Makefile src/Makefile include/Makefile example/Makefile)
```

example/Makefile.am

```
LDFLAGS = -lXterminal
bin_PROGRAMS = exapp exbar exdlg exevent exfreeapp exhlist exlist \
               exmyapp extab exwin
exapp_SOURCES = exapp.cc
exbar_SOURCES = exbar.cc
exdlg_SOURCES = exdlg.cc
```



```
exevent_SOURCES = exevent.cc
exfreeapp_SOURCES = exfreeapp.cc
exhlist_SOURCES = exhlist.cc
exlist_SOURCES = exlist.cc
exmyapp_SOURCES = exmyapp.cc
extab_SOURCES = extab.cc
exwin_SOURCES = exwin.cc
```

include/Makefile.am

```
include_HEADERS = xbclass.h xconfig.h xevent.h xobject.h xtapp.h \
                  xtbar.h xtbutton.h xtcombobox.h xtdesktop.h \
                  xtdialog.h xterminal.h xtfield.h xthlist.h xtlib.h \
                  xtlist.h xtmenu.h xtother.h xtpassive.h xtscreen.h \
                  xttab.h xttexteditor.h xttools.h xtwindow.h \
                  RegionalInfo.h

uninstall-local:
    rmdir $(includedir)
```

xconfig.h.in

```
/*  include/xconfig.h.in.  Generated  automatically  from
configure.in by autoheader.  */

/* Define if you don't have vprintf but do have _doprnt.  */
#undef HAVE_DOPRNT

/* Define if you have the vprintf function.  */
#undef HAVE_VPRINTF

/* Define as the return type of signal handlers (int or void).  */
#undef RETSIGTYPE

/* Define if you have the ANSI C header files.  */
#undef STDC_HEADERS

/* Define if your declares struct tm.  */
#undef TM_IN_SYS_TIME

/* Define if you want to catch the signals and translate them into
events. */
#undef SIGNAL_CATCHING

/* Define if you don't want to use that ugly ncurses hacks. */
#undef NO_NCURSES_HACKS

/* Define if you want to generate debugging informations. */
#undef DEBUG
```

```
/* Define if you are using `ncurses' as curses library. */
#undef HAVE_NCURSES

/* Define if you are using `slang' as curses library. */
#undef HAVE_SLANG

/* Define if you want to use the Linux console mouse support. */
#undef HAVE_LIBGPM

/* Define if your curses have `ESCDELAY'. */
#undef HAVE_ESCDELAY

/* Define if you have the resizeterm function.  */
#undef HAVE_RESIZETERM

/* Define if you have the strdup function.  */
#undef HAVE_STRDUP

/* Define if you have the strtod function.  */
#undef HAVE_STRTOD

/* Define if you have the <string.h> header file.  */
#undef HAVE_STRING_H

/* Define if you have the <strings.h> header file.  */
#undef HAVE_STRINGS_H

/* Define if you have the <sys/ioctl.h> header file.  */
#undef HAVE_SYS_IOCTL_H

/* Define if you have the <unistd.h> header file.  */
#undef HAVE_UNISTD_H

/* Name of package */
#undef PACKAGE

/* Version number of package */
#undef VERSION

/* Define if TIOCGWINSZ requires sys/ioctl.h */
#undef GWINSZ_IN_SYS_IOCTL
```

src/Makefile.am

```
lib_LTLIBRARIES = libXterminal.la
libXterminal_la_SOURCES = xbclass.cc xevent.cc xobject.cc xtapp.cc \
                          xtbar.cc xtbutton.cc xtcombobox.cc \
                          xtdesktop.cc xtdialog.cc xtfield.cc \
                          xthlist.cc xtlib.cc xtlist.cc xtmenu.cc \
                          xtother.cc xtpassive.cc xtscreen.cc xttab.cc \
                          xttexteditor.cc xttools.cc xtwindow.cc \
                          RegionalInfo.cc
```



```
libXterminal_la_LDFLAGS = -version-info @CURRENT@:@REVISION@:@AGE@
```

Mai multe informaţii despre `autoconf`, `automake` şi `libtool` pot fi obţinute din manualul `info` pentru aceste utilitare.

3.3.1. Asigurarea portabilităţii programelor

Portabilitatea este o calitate a codului-sursă care îi permite să fie compilat şi executat pe o varietate de platforme. În contextul cărţii de faţă, portabilitatea se referă în general la calitatea de a rula pe diferite sisteme compatibile UNIX, uneori şi Windows.

Prezentăm în cele ce urmează o serie de idei care trebuie urmate în vederea asigurării unui nivel de portabilitate cât mai ridicat.

1. Trebuie utilizate funcţii de bibliotecă conforme specificaţiilor ANSI (vezi Anexa 1) şi trebuie evitate funcţiile specifice unei anumite platforme.
2. Dacă trebuie folosite funcţii specifice unei anumite platforme sau biblioteci, existenţa acestora trebuie testată în cadrul scriptului `configure`. Codul va trebui să includă fişierul-antet `config.h` generat şi să efectueze teste prin intermediul directivelor preprocesor ca în exemplul de mai jos:

```
#ifdef HAVE_NCURSES
strcpy(s, termname());
#else
strcpy(s, getenv("TERM"));
#endif // HAVE_NCURSES
```

3. De asemenea, anumite fişiere-antet au denumiri diferite pe diferite platforme:

```
#ifdef HAVE_STRING_H
#include <string.h>
#else
#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif
#endif
```

4. Este indicat includerea în toate fişierele-antet a câtorva directive preprocesor pentru a preveni includerea multiplă:

```
#ifndef FUNCTII_H
#define FUNCTII_H 1
...
#endif /* FUNCTII_H */
```

5. Pentru ca un program C++ să poată utiliza o bibliotecă compilată cu un compilator C, este necesar ca orice simbol exportat să fie declarat `extern „C"`:

```
#ifdef __cplusplus
extern "C" {
#endif
...
#ifdef __cplusplus
}
#endif
```

3.4. Utilitarele *diff, patch* şi *diffutils*

3.4.1. Utilitarul *diff*

Utilitarul `diff` determină diferenţele dintre două fişiere text. Acest program este util pentru a evidenţia modificările aduse unui program-sursă cu ocazia unei versiuni mai noi, sau cele efectuate de diferiţi programatori. Programul `diff` a fost scris de către Mike Haertel, David Hayes, Richard Stallman, Len Tower, Paul Eggert şi Wayne Davison, prima versiune datând din anul 1992.

Acest program se găseşte în pachetul `difftutils`.

`diff` trimite la ieşirea standard rezultatul comparaţiei, care poate fi codificat utilizând mai multe formate (vezi *infra*). Setul de diferenţe generat este denumit de obicei *diff* sau *patch*. Dacă fişierele sunt identice, nu este trimis nici un rezultat la ieşire.

Setul de diferenţe generat poate fi utilizat pentru a distribui modificări (sau *update*-uri) ale programelor-sursă către alte persoane. Această metodă este utilă atunci când diferenţele sunt mici în comparaţie cu fişierele-sursă. De asemenea, `diff` poate compara fişierele aflate pe două structuri de directoare (*source trees*). Pentru a *„aplica"* setul de diferenţe unui fişier *„origine"* în vederea generării fişierului modificat, se utilizează utilitarul `patch`.

Spre exemplu, comanda `diff` *`fisier1_original fisier_modificat`* generează setul de diferenţe dintre cele două fişiere specificate în linia de comandă.

După cum afirmam mai sus, pot fi utilizate mai multe formate pentru ieşirea generată. Varianta implicită, numită *„normală"*, include doar liniile modificate, şterse sau adăugate. Pentru exemplificare, să considerăm următoarele două fişiere-sursă:

hello.c:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");

    return 0;
}
```

hello_ro.c:

```
#include <stdio.h>

int main()
{
    printf("Salut!\n");

    return 0;
}
```

Iată un exemplu de *patch* generat în urma execuţiei comenzii `diff hello.c hello_ro.c`.

```
5c5
<printf("Hello, world!\n");
---
>printf("Salut!\n");
```

În mod uzual, atunci când studiem diferenţele dintre două fişiere, dorim să vedem liniile de text apropiate de cele care au fost modificate, pentru a înţelege exact ce a fost schimbat. Aceste linii de text apropiate sunt denumite *„context"*. Programul `diff` oferă două formate pentru setul de diferenţe care includ contextul modificărilor: formatul *„context"* şi formatul *unificat*. Folosirea acestor două formate mai are avantajul că programul `patch` poate aplica modificările efectuate chiar dacă s-au efectuat modificări ale fişierului original (*e.g.* s-au adăugat linii de text).

Generarea setului de modificări în format context: `diff -c` *`fisier_origine fisier_modificat`*. Prezentăm mai jos un exemplu:

```
*** hello.c Tue Jul 23 12:56:49 2002
--- hello_ro.cTue Jul 23 12:57:09 2002
***************
*** 2,8 ****

 int main()
 {
!   printf("Hello, world!\n");

    return 0;
```

```
 }
--- 2,8 ----

 int main()
 {
!   printf("Salut!\n");

    return 0;
 }
```

Caracterul de început al liniilor de text modificate poate fi:

- „!": o linie de text care face parte dintr-un grup de una sau mai multe linii care au fost modificate;
- „+": o linie de text introdusă;
- „-": o linie de text ştearsă.

Formatul *diff* unificat este mai compact decât cel context. Generarea se face cu: `diff -u fisier_origine fisier_modificat`. Iată un exemplu:

```
--- hello.c Tue Jul 23 12:56:49 2002
+++ hello_ro.cTue Jul 23 12:57:09 2002
@@ -2,7 +2,7 @@

 int main()
 {
-   printf("Hello, world!\n");
+   printf("Salut!\n");

    return 0;
 }
```

Caracterul de început al liniilor de text modificate poate fi:

- „+": o linie de text introdusă;
- „-": o linie de text ştearsă.

Pentru ca diferenţele listate de comanda `diff` să fie memorate într-un fişier în loc să fie afişate pe ecran, poate fi utilizată comanda de redirectare a ieşirii – „>" –, specificându-se numele noului fişier (de tip *patch*) care va fi creat:

```
diff [ optiuni ] fisier_origine fisier_modificat > fisier_patch
```

Alţi parametri utili ai comenzii `diff` sunt următorii:

- `-q` raportează doar faptul că un fişier a fost modificat, nu şi detaliile modificării;
- `-r` parcurge recursiv directoarele, atunci când se face comparaţia între două directoare.

Mai multe informaţii despre utilitarul `diff` pot fi obţinute apelând comanda `info diff`.

3.4.2. Utilitarul *patch*

Pentru a aplica un fişier *diff* unui fişier sau arbore de fişiere original, se utilizează utilitarul `patch`. Acesta a fost implementat pentru prima oară în anul 1992 de către Larry Wall şi Paul Eggert.

Acest program se găseşte în pachetul `difftutils`.

Numele fişierelor care vor fi modificate sunt în mod uzual preluate din fişierul *diff*. De asemenea, formatul utilizat pentru *patch* este determinat în mod automat. Sintaxa simplificată este `patch < fisier_patch`.

Parametrii mai importanţi sunt:

- `-b` creează fişiere de siguranţă *backup;*
- `-c` consideră că fişierul *patch* este în format context;
- `-n` consideră că fişierul de intrare este în format normal;
- `-pnumar` elimină căile de directoare conţinând numărul de niveluri (caractere „/") specificat din numele fişierelor. Acest parametru este util atunci când fişierele sunt stocate într-o ierarhie de directoare la persoana care a generat *patch*-ul şi în alt mod la persoana care îl utilizează. Spre exemplu, dacă numele fişierelor din *patch* este `/home/dragos/work/doc++/src/doc2xml.ll`, utilizând parametrul `-p3`, acesta va fi transformat în `doc++/src/doc2xml.ll`;
- `-R` consideră că *patch*-ul a fost creat cu fişierele original respectiv modificat schimbate între ele;
- `-u` consideră formatul fişierului de intrare ca fiind *diff* unificat.

Mai multe informaţii despre utilitarul `patch` pot fi obţinute apelând comanda `info patch`.

3.4.3. Utilitarul *diffstat*

Utilitarul `diffstat` realizează o analiză statistică a unui fişier *patch*, şi anume a numărului de inserări, ştergeri şi modificări ale tuturor fişierelor incluse în *patch*. Spre exemplu, comanda `diffstat patch-3.4.10-pre1.gz` va afişa o statistică a fişierului *patch* specificat (`diffstat` va decomprima automat fişierul):

```
CREDITS     |  2
README      |  15
REPORTING-BUGS  |  6
doc/ChangeLog   |  13
```

```
 po/fr.po      | 4
 po/ja.po      | 4
 po/ro.po      | 6
 po/ru.po      | 4
 src/cpp.ll    |  10
 src/doc2db.ll | 1594 ------------------------------
 src/doc2dbsgml.ll |  1594 ++++++++++++++++++++++++++++++
 src/doc2dbxml.ll  |  1656 ++++++++++++++++++++++++++++++++
 src/equate.cc | 4
 src/main.cc   |  32
 14 files changed, 3250 insertions(+), 1594 deletions(-)
```

3.5. Sistemul de control al versiunilor CVS

CVS (*Concurrent Versioning System*) este un sistem de control al versiunii, care permite păstrarea tuturor versiunilor unor fişiere (în general cod-sursă) şi realizarea unui jurnal cu modificările efectuate de diferiţi utilizatori. Acest sistem permite ca un grup de persoane (echipă) să lucreze la un grup de fişiere (proiect) comun. Sistemul CVS a fost creat în 1989 de către Brian Berliner şi Jeff Polk, pe baza unor scripturi shell realizate în 1986 de către Dick Grune.

Pentru a putea utiliza CVS trebuie instalat pachetul `cvs`.

Sistemul consistă dintr-un *depozit* (*repository*), stocat pe un server, care conţine cea mai recentă versiune a grupului de fişiere. Utilizatorii se pot conecta în orice moment la acest depozit şi pot prelua fişierele pe care le conţine, sau îşi pot actualiza o versiune mai veche, existentă pe discul local. Fiecare utilizator efectuează modificări asupra fişierelor şi, în momentul în care doreşte, trimite modificările către depozit.

O sesiune CVS începe aşadar cu obţinerea unei copii a arborelui conţinând fişierele-sursă ale proiectului `doc++`:

```
cvs checkout doc++
```

Această comandă va crea un director `doc++` în directorul curent care va conţine programele-sursă ale proiectului menţionat mai sus. În directorul nou creat precum şi în subdirectoarele componente există câte un director numit `CVS`. În mod normal, conţinutul acestui director nu trebuie modificat.

Pot fi modificate fişiere-sursă prin intermediul unui editor, după care aceste modificări pot fi trimise (*committed*) către depozitul CVS utilizând comanda:

```
cvs commit
```


Sistemul CVS va lansa un editor care va solicita introducerea unui mesaj ce va fi introdus în jurnalul de operaţiuni. În mod uzual mesajul trebuie să conţină o descriere a modificărilor efectuate. Programul care va fi apelat ca editor se stabileşte prin intermediul variabilei de mediu `CVSEDITOR`. Pentru a se evita execuţia editorului poate fi utilizată comanda (care va actualiza doar un singur fişier care a fost modificat):

```
cvs commit -m "Optimizat algoritmul de sortare" src/cpp.ll
```

Pentru a actualiza copia locală a fişierelor-sursă, se foloseşte comanda:

```
cvs update
```

Pentru a adăuga un fişier nou creat şi în depozitul CVS, se utilizează comanda:

```
cvs add src/doc2dbxml.ll
```

Pentru a şterge un fişier eliminat din arborele local şi din depozitul CVS, se utilizează comanda:

```
cvs remove src/doc2db.ll
```

Pentru a vizualiza diferenţele dintre un fişier local (original) şi un fişier aflat în CVS (modificat):

```
cvs diff doc/ChangeLog
```

Ieşirea generată este în format *diff* unificat (vezi *supra*).

Alte opţiuni utile ale comenzii `cvs`:

- `-d` *`director`* specifică locaţia depozitului. *director* poate avea şi formatul *utilizator@maşină:director*. Parametrul opţiunii `-d` poate fi specificat şi prin intermediul variabilei de mediu `CVSROOT`. Spre exemplu, un director local va fi specificat ca `/usr/local/cvsroot`, iar unul aflat pe o altă maşină `adragos@cvs.docpp.sourceforge.net:/ cvsroot/docpp`.

Informaţii detaliate privind utilizarea serviciului CVS pot fi obţinute apelând comanda `info cvs`.

Pentru a utiliza sistemul CVS poate fi folosit şi programul `Cervisia` (meniul `Development :: Cervisia (CVS Frontend)`):

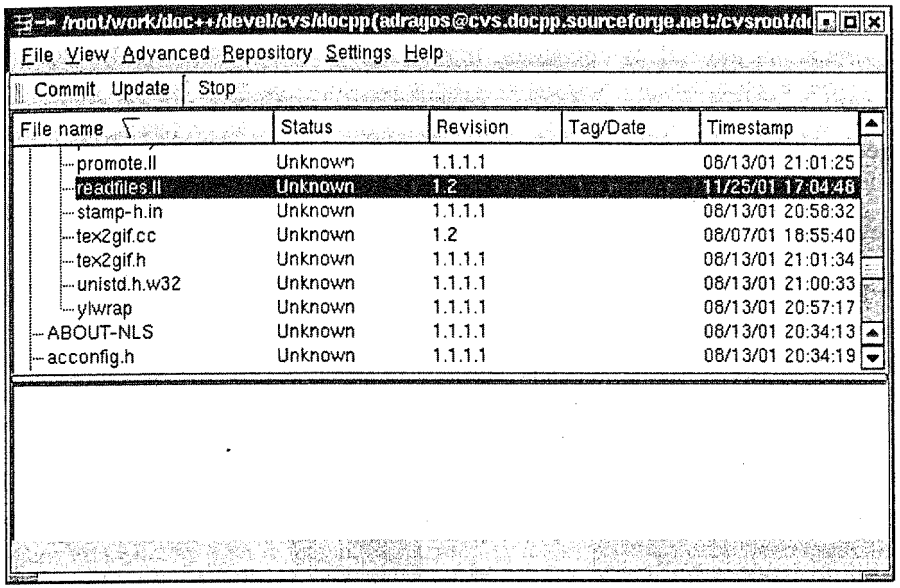


Figura 3.1. Programul Cervisia

3.5.1. Configurarea unui server CVS

Această secţiune prezintă modul de configurare a unui calculator care va stoca un depozit CVS şi va servi cereri CVS.

Pentru a configura CVS ca server, trebuie modificat în primul rând fişierul `/etc/inetd.conf`, pentru ca `inetd` să execute comanda `cvs pserver` atunci când recepţionează o cerere de conexiune pe portul 2401 (portul standard CVS):

```
2401  stream  tcp  nowait       root     /usr/bin/cvs
cvs -f --allow-root=/var/lib/cvsroot/docpp pserver
```

Opţiunea `--allow-root` specifică directorul CVSROOT permis. Acest director se referă la directorul în care se află stocat un anumit depozit, în exemplul nostru `/var/lib/cvsroot/docpp`. Directorul se stabileşte de către administratorul sistemului. Dacă există mai multe depozite la care se doreşte permiterea accesului, opţiunea trebuie repetată.

Dacă se doreşte referirea portului ca un nume simbolic de serviciu, se poate adăuga în fişierul `/etc/services` liniile următoare:

```
cvspserver  2401/tcp
cvspserver  2401/udp
```


După modificarea fişierului `/etc/inetd.conf`, daemonul `inetd` trebuie repornit.

Sistemele mai noi oferă, ca alternativă la `inetd`, daemonul `xinetd`. Pentru configurarea serverului CVS este necesară crearea unui fişier denumit `cvspserver` în directorul `/etc/xinetd.d`, cu următorul conţinut:

```
service cvspserver
{
 disable         = no
 socket_type     = stream
 wait            = no
 protocol        = tcp
 port            = 2401
 user            = root
 server          = /usr/bin/cvs
 server_args     = -f --allow-root=/var/lib/cvsroot/docpp pserver
 log_on_success  += USERID
 log_on_failure  += USERID
}
```

Pentru mai multe detalii privind configurarea serviciilor `inetd` recomandăm consultarea lucrării Dragoş Acostăchioaie, *Administrarea şi configurarea sistemelor Linux*, Editura Polirom, Iaşi, 2002.

Pentru autentificarea utilizatorilor se foloseşte în general un fişier separat de parole, astfel încât utilizatorii nu vor folosi parola lor obişnuită de intrare în sistem. Fişierul de parole CVS este `$CVSROOT/CVSROOT/passwd`, care foloseşte un sistem similar cu `/etc/passwd`, exceptând faptul că include mai puţine câmpuri: numele de utilizator CVS, parola (opţional) şi un nume de utilizator sub care va rula CVS (opţional). Parolele sunt criptate utilizând mecanismul standard `crypt`. Iată un fişier exemplu:

```
anonymous:
dragos:weknEDYU47q
mihaela:37cLKDuee:pubcvs
```

Prima linie din fişier permite accesul oricărui client CVS care încearcă să se autentifice cu utilizatorul `anonymous`, fără a solicita vreo parolă (acest gen de utilizator este folosit de obicei pentru a oferi acces doar pentru citire – read-only – la depozitul CVS).

A treia linie va acorda drept de acces utilizatorului `mihaela`, dar operaţiile CVS vor fi executate pe server sub utilizatorul `pubcvs`. Nu este necesar ca utilizatorul `mihaela` să existe în sistem, dar utilizatorul cu numele `pubcvs` trebuie să fie definit. Aceasta permite definirea de mai mulţi utilizatori CVS fără a fi necesar definirea acestora ca utilizatori ai sistemului.

CVS poate folosi şi autentificarea oferită de sistemul de operare. Astfel, dacă utilizatorul specificat în `$CVSROOT/CVSROOT/passwd` nu este găsit, sau dacă acest fişier de parole nu există, serverul va presupune că utilizatorul există în sistem şi va încerca autentificarea acestuia.

În mod evident, pentru a asigura securitatea depozitului CVS, trebuie corect configurate drepturile de acces ale fişierelor şi subdirectoarelor din cadrul directorului `$CVSROOT/CVSROOT`.

Pentru a putea stabili care utilizatori au permisiune de scriere în depozitul CVS, sau care au doar permisiune de citire, sunt utilizate două fişiere. Primul fişier, `$CVSROOT/CVSROOT/readers`, declară utilizatorii care au dreptul de citire, câte unul pe fiecare linie, ca în exemplul de mai jos:

```
anonymous
dragos
mihaela
```

Într-un mod similar, fişierul `$CVSROOT/CVSROOT/writers` defineşte utilizatorii care au drept de scriere:

```
dragos
mihaela
```

3.6. Alte programe utile

3.6.1. Programul *indent*

Programul `indent` se utilizează pentru a formata programul-sursă C/C++ după un anumit stil. Utilizarea unui stil unitar conduce la o lizibilitate mai mare a codului şi, ca urmare, la un cost de mentenanţă mai redus.

Sintaxa generală este:

```
indent [ optiuni ] fisiere_sursa
```

Dacă nu este specificată nici o opţiune, fişierele-sursă vor fi formatate conform stilului GNU (vezi documentul *The GNU Coding Standards*: `http://www.gnu.org/prep/standards.html`). Pentru a formata codul-sursă conform recomandărilor Kerningham & Ritchie (vezi *The C Programming Language*), se poate utiliza opţiunea `-kr`.

Opţiunile mai importante (grupate după categoria acestora) sunt:

Linii vide

- `-bad` introduce o linie vidă după fiecare bloc de declaraţii;
- `-bap` introduce o linie vidă după fiecare funcţie;
- `-bbb` introduce o linie vidă înainte de fiecare bloc de comentarii.

Comentarii

`indent` formatează atât comentariile C (`/* ... */`), cât şi cele C++ (`// ...`). Comentariile de tip bloc (sau cutie) nu sunt modificate:

```
/************************
 *   comentariu   *
 ************************/
```

sau:

```
/*
 * comentariu
 */
```

Marginea din dreapta (practic numărul de caractere pe o linie) pentru comentarii este implicit 78, această valoare putând fi modificată prin intermediul opţiunii `-lc`. Aceasta poate fi utilă spre exemplu pentru a putea tipări codul-sursă.

Dacă linia pe care se află comentariul nu conţine cod-sursă, acesta va fi aliniat la coloana pe care este aliniat şi codul-sursă din blocul curent. Acest stil de aliniere poate fi modificat cu ajutorul opţiunii `-d`, care specifică numărul de caractere cu care comentariul va fi mutat la stânga.

Dacă linia pe care se află comentariul conţine cod-sursă, comentariile vor fi aliniate implicit la coloana 33. Această valoare poate fi modificată cu ajutorul următoarelor trei opţiuni:

- `-c` specifică coloana de aliniere pentru comentariile care urmează după cod;
- `-cd` specifică coloana de aliniere pentru comentariile care urmează după declaraţii;
- `-cp` stabileşte coloana de aliniere pentru comentariile care urmează după directive ale preprocesorului.

Opţiunea `-cdb` mută delimitatorii comentariilor pe linii separate. Astfel, un comentariu de genul `/* comentariu */` va fi transformat în:

```
/*
   comentariu
 */
```

Opţiunea `-sc` va introduce un caracter „*" la începutul fiecărei linii dintr-un comentariu. Astfel, prin utilizarea opţiunilor combinate `-cdb -sc`, vom obţine:

```
/*
 * comentariu
 */
```

Instrucţiuni

Opţiunea `-br` formatează acoladele astfel:

```
if(i == 0) {
    i++;
}
```

Opţiunea `-bl` formatează acoladele astfel:

```
if(i == 0)
{
    i++;
}
```

Numărul de spaţii cu care acoladele sunt indentate poate fi modificat cu opţiunea `-bli`. În exemplul de mai sus s-a folosit opţiunea `-bli4`.

Indentarea etichetelor `case` într-o construcţie `switch` se poate face cu ajutorul opţiunii `-cliN`. Indentarea acoladelor de sub etichetele `case` se modifică cu opţiunea `-cbiN`.

În urma folosirii opţiunii `-pcs` va fi introdus un spaţiu după numele funcţiilor şi caracterul „(" la apelarea acestora, cum ar fi `printf ("text\n");`. Opţiunea `-cs` introduce un spaţiu după un operator cast, iar `-bs` între cuvântul `sizeof` şi argumentul său.

Opţiunea `-saf` introduce un spaţiu între `for` şi paranteza care urmează. `-sai` inserează un spaţiu între `if` şi paranteza care urmează. `-saw` introduce un spaţiu după instrucţiunea `while`.

Opţiunea `-prs` separă printr-un spaţiu parantezele de conţinutul dintre ele.

Declaraţii

Identificatorii pot fi aliniaţi la coloana specificată cu opţiunea `-di`. Spre exemplu, opţiunea `-di8` va formata declaraţiile ca mai jos:

```
int i;
char  *s;
```


Argumentele unei funcţii pot fi aranjate câte unul pe fiecare linie utilizând opţiunea `-bfda`, util pentru funcţii cu număr mare de argumente. Astfel, funcţia:

```
int main(int argc, char **argv)
```

va fi formatată ca:

```
int main(int argc,
         char **argv)
```

Poziţia acoladelor din cadrul declaraţiilor `struct` poate fi modificată cu ajutorul opţiunii `-brs`:

```
struct student {
    char *nume;
    char *prenume;
};
```

Opţiunea `-bls` produce următorul format:

```
struct student
{
    char *nume;
    char *prenume;
};
```

Indentarea liniilor de cod

Atunci când în cadrul codului-sursă apar instrucţiuni de tipul `if` sau `for`, nivelul de indentare este incrementat cu valoarea specificată de opţiunea `-i`. Spre exemplu, `-i8` va indenta codul-sursă cu opt caractere. Atunci când o expresie ocupă mai multe linii, a doua linie şi următoarele sunt indentate cu un număr suplimentar de spaţii, specificat cu ajutorul opţiunii `-ci` (implicit 0).

Despărţirea liniilor lungi

Lungimea maximă a unei linii de cod poate fi stabilită utilizând opţiunea `-l`. Aceasta nu include comentariile care urmează după codul efectiv.

Dezactivarea formatării automate

Programul `indent` permite dezactivarea formatării pe o porţiune din codul-sursă. Pentru a realiza acest lucru, trebuie plasat comentariul de control `/* *INDENT-OFF* */` (sau echivalentul C++ `// *INDENT-OFF*`), comentariu după care codul nu va fi formatat. Această dezactivare este activă până la sfârşitul fişierului-sursă sau până la întâlnirea comentariului `/* *INDENT-ON* */`.

Informaţii suplimentare despre `indent` pot fi obţinute consultând pagina `info indent`.

3.6.2. Alte utilitare

Alte utilitare folosite sunt cele din pachetele `binutils` şi `fileutils`:

- `strip` – elimină toate simbolurile dintr-un fişier-obiect. Acestea includ simbolurile pentru depanare precum şi altele care nu sunt utile pentru utilizatorul final al executabilului;
- `strings` – afişează toate şirurile de caractere conţinute de un fişier-obiect;
- `touch` – modifică timpul de acces sau modificare – implicit timpul curent – a unor fişiere.

3.7. Documentarea programelor: *DOC++* şi *code2html*

3.7.1. *DOC++*

DOC++ este un generator de documentaţii pentru C şi C++, producând ieşire atât în format TeX, pentru a putea genera versiuni tipărite ale documentaţiei, cât şi HTML pentru navigare uşoară prin documentaţie. Documentaţia este extrasă direct din fişierele-antet sau sursă C/C++. Programul DOC++ a fost conceput în anul 1995 de către Roland Wunderling şi Malte Zöckler, dezvoltarea acestuia fiind continuată începând cu 1998 de către Dragoş Acostăchioaie (autorul cărţii de faţă).

Acest utilitar poate fi descărcat de pe Internet de la adresa `http://docpp.sourceforge.net/download.html`. Pachetul care conţine programul se numeşte `doc++`.

Programul DOC++ permite programatorilor să scrie documentaţie, putându-se concentra în acelaşi timp pe dezvoltarea programelor. Pentru a realiza acest deziderat, documentaţia este localizată chiar în cadrul codului-sursă al proiectului pe care îl dezvoltă. Urmând un astfel de model, un programator poate scrie documentaţie pentru clasele, funcţiile sale etc. şi să o actualizeze în funcţie de modificările aduse codului-sursă.

Pentru a putea face distincţie între comentariile obişnuite, legate în general de implementare, DOC++ utilizează nişte tipuri speciale de comentarii, care au formatul:

```
/** ... */
/// ...
```


Vom numi un asemenea comentariu, *comentariu DOC++*. Fiecare dintre acestea se utilizează pentru a specifica documentaţia pentru declaraţia care urmează.

Fiecare comentariu DOC++ defineşte o *intrare în manual*, care constă în documentaţia furnizată de comentariul în sine şi diferite informaţii din declaraţia care urmează. Intrările în manual sunt structurate în mai multe câmpuri: unele dintre ele sunt completate în mod automat de către DOC++, iar altele pot fi specificate de către programator:

Tabelul 3.1. Intrări în manuale DOC++

Numele câmpului	Furnizor	Descriere
`@type`	DOC++	tipul (returnat de) obiect
`@name`	DOC++, programator	numele obiectului
`@args`	DOC++	argumentele primite de obiect
`@memo`	programator	documentaţia pe scurt
`@doc`	programator	documentaţia completă
`@return`	programator	documentaţia pentru valoarea returnată de funcţie
`@param`	programator	documentaţia pentru un argument al funcţiei
`@exception`	programator	documentaţia pentru o excepţie interceptată
`@see`	programator	referinţă
`@author`	programator	autorul
`@version`	programator	versiunea

DOC++ permite gruparea intrărilor în manual, cu ajutorul construcţiei:

```
/** @name numele_grupului
    (documentatia pentru grup)
*/
//@{
    (alte intrari in manual)
//@}
```

Va fi creat un grup având numele specificat, care conţine toate celelalte intrări ca sub-intrări.

Există posibilitatea includerii altor fişiere conţinând documentaţie, folosind sintaxa:

```
//@Include: lista_fisiere
/*@Include: lista_fisiere */
```

Pentru formatarea documentaţiei pot fi utilizate atât macrouri (tag-uri) de formatare HTML, cât şi LaTeX, ele fiind convertite în mod automat la tipul ieşirii generate.

```
/** O clasa de baza oarecare.
    Documentatia pentru aceasta clasa. A se vedea si situl
    proiectului de la adresa \URL{http://commonbase.sourceforge.net}
*/
class CommonBase {
private:
    /**acest membru va fi listat in documentatie doar daca
       utilizam optiunea `--private'
    */
    int privateMember();

protected:
    /// o variabila oarecare
    double variable;

public:
    /// o functie membra publica
    int publicMember();
};

/** O functie globala
    Aceasta este documentatia functiei

    @param c caracterul de intrare
    @return 0 daca totul este OK
    @return -1 in caz de eroare
    @author Dragos Acostachioaie
    @version 5.7.1
    @see CommonBase
*/
int function(const int c);
```

Pentru a genera documentaţia, vom apela programul `doc++`, astfel:

```
doc++ --dir html lista_fisiere
doc++ --tex --output doc.tex lista_fisiere
```

Prima variantă generează documentaţia în format HTML, iar a doua în format LaTeX.

Alte opţiuni utile sunt:

- `-A` generează intrări de manual pentru toate declaraţiile din codul-sursă, chiar dacă acestea nu sunt documentate;
- `-c` utilizează comentariile C/C++ ca fiind comentarii DOC++;
- `-p` include şi membrii privaţi în documentaţie.

Pentru mai multe informaţii privind utilizarea `DOC++`, recomandăm consultarea documentaţiei localizate în `/usr/share/doc/doc++-versiune/manual/html/`.

3.7.2. *code2html*

Utilitarul `code2html` converteşte fişiere-sursă C şi C++ în cod HTML, folosind mecanismul numit *syntax highlighting*, altfel spus aplică diferite scheme de colorare pentru instrucţiuni, paranteze etc. Programul `code2html` este scris în Perl şi a fost dezvoltat începând cu anul 1999 de către Peter Palfrader.

Programul `code2html` poate fi descărcat de la adresa `http://code2html.sourceforge.net`.

Cel mai simplu exemplu de utilizare a acestui program este:

```
code2html atoi.c out.html
```

care generează fişierul HTML corespunzător programului C `atoi.c`.

Alte opţiuni des utilizate sunt următoarele:

- `-l` specifică limbajul în care este scris fişierul-sursă (util în cazul în care `code2html` nu poate determina automat tipul acestuia). Sunt recunoscute „c" şi „c++" (precum şi o serie de alte limbaje);
- `-n` inserează şi numărul liniei din codul-sursă.

Alte programe pentru documentare automată:

- `doxygen`: http://doxygen.sourceforge.net;
- `cxx2html`: http://aips2.aoc.nrao.edu/RELEASED/cxx2html.

3.8. Verificarea programelor: *splint*

Programul `splint` este un instrument pentru verificarea programelor C. El poate descoperi erori de programare şi posibile vulnerabilităţi cum ar fi pointeri utilizaţi înainte de a fi alocaţi, depăşirea dimensiunii maxime a şirurilor de caractere, cicluri infinite etc. Programul nu verifică corectitudinea sintaxei C, ci erorile de programare. Aceste erori nu pot fi întotdeauna detectate de către programator, nici chiar în urma depanării programelor, deoarece programele nu pot fi testate în toate condiţiile posibile. Utilitarul `splint` (*Secure Programming Lint*) este succesorul programului `LCLint`, şi este dezvoltat de *Secure Programming Group*, condus de către David Evans, în cadrul Universităţii din Virginia.

Programul `splint` se găseşte în pachetul cu acelaşi nume.

splint primeşte în mod uzual ca argument lista fişierelor C care vor fi verificate. Spre exemplu:

```
# splint test.c
Splint 3.0.1.6 --- 26 Feb 2002

test.c: (in function main)
test.c:5:12: Unallocated storage s passed as out parameter to
strcpy: s
An rvalue is used that may not be initialized to a value on some
execution path. (Use -usedef to inhibit warning)
test.c:6:2: Path with no return in function declared to return int
There is a path through a function declared to return a value on
which there is no return statement. This means the execution may
fall through without returning a meaningful result to the caller.
(Use -noret to inhibit warning)

Finished checking --- 2 code warnings
```

După cum se poate observa, splint a descoperit două erori:

1. Am folosit la apelul funcţiei strcpy argumentul s, un şir de caractere pentru care nu am alocat spaţiu de memorie. Cu alte cuvinte, apelul strcpy va avea rezultate imprevizibile (în general va duce la terminarea forţată a programului).
2. Funcţia main trebuie să returneze un int, dar noi nu am returnat explicit nici o valoare. Altfel spus, valoarea returnată în urma execuţiei programului este incertă.

Pentru mai multe informaţii privind splint, recomandăm manualul de utilizare, localizat în genere în /usr/share/doc/splint-*versiune*/manual.html.

Alt program care poate fi folosit cu succes (dar nu atât de evoluat ca splint) este clint.

Capitolul 4

MEDII INTEGRATE DE DEZVOLTARE

Mulţi oameni, incluzând mulţi programatori,
profesori şi manageri, refuză pur şi simplu să înţeleagă
complexitatea dezvoltării de programe.
(Bjarne Stroupstrup)

4.1. Depanatorul vizual *DDD*

DDD (*Data Display Debugger*) este un aşa-zis *debugger vizual*, cu alte cuvinte o interfaţă grafică intuitivă pentru programul de depanare GDB. DDD dispune atât de interfaţă în mod terminal, cât şi în mediu grafic X Window. Autorii originali ai DDD sunt Dorothea Lütkehaus şi Andreas Zeller, care au început dezvoltarea acestuia în anul 1995.

Acest program este localizat în pachetul ddd.

DDD se lansează în execuţie prin comanda ddd. Ecranul DDD conţine trei ferestre, şi anume cea pentru inspectarea variabilelor (în partea de sus), cea pentru programul-sursă şi cea pentru consola pentru mesajele GDB. Programul conţine şi un panou de comandă, conţinând funcţiile mai des utilizate.

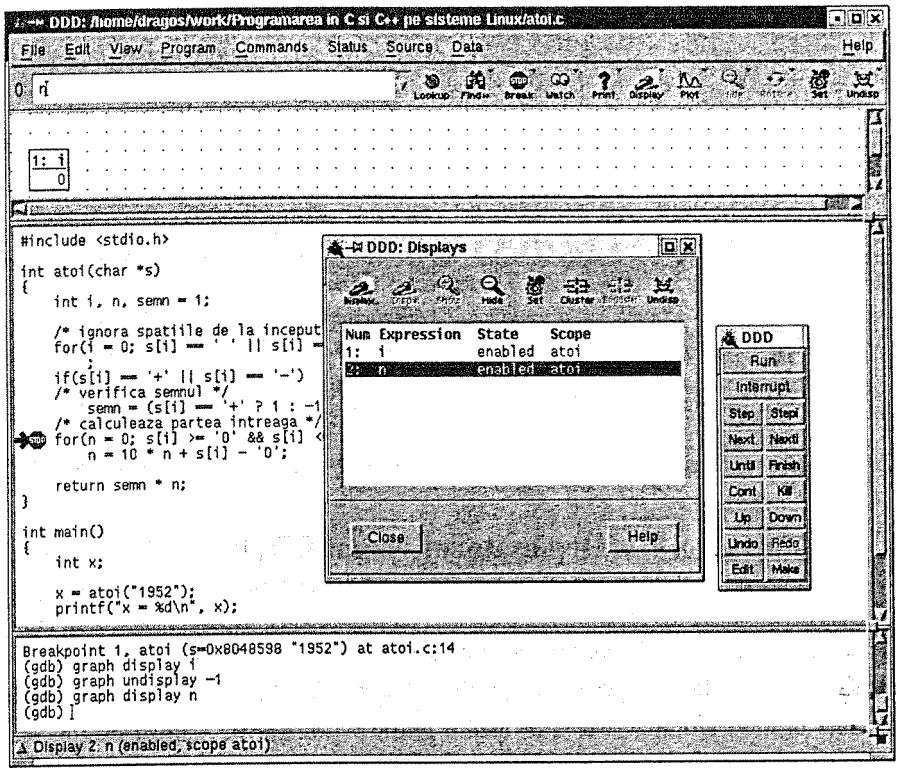


Figura 4.1. Programul DDD

Pot fi gestionate cu uşurinţă punctele de oprire, utilizând tehnologia *drag-and-drop*. Acestea pot fi salvate pentru sesiuni viitoare de lucru. Programul oferă modalităţi avansate de inspectare a variabilelor, în special a structurilor şi tablourilor.

4.2. Mediul integrat *KDevelop*

`KDevelop` este un mediu integrat de dezvoltare (*IDE*) pentru C şi C++. El include un editor de texte, o interfaţă pentru depanatorul `GDB`, un manager de proiect şi un browser sofisticat pentru sursele programelor (browser de clase C++ etc.). `KDevelop` permite dezvoltarea uşoară de aplicaţii `KDE` bazate pe biblioteca `Qt`, cu interfeţe grafice performante. De asemenea, programul include şi suport pentru `CVS`. Programul `KDevelop` a fost conceput de către Sandy Meier, prima versiune operaţională fiind lansată în anul 1998.

Pentru a utiliza acest program trebuie ca pachetul `kdevelop` să fie instalat. Se lansează în execuţie prin comanda cu acelaşi nume.

`KDevelop` conţine *vrăjitori* (*wizards*) care permit generarea automată a scheletului proiectului, a structurii de directoare a acestuia, a fişierelor de configurare `autoconf` şi `automake`, în funcţie de interfaţa proiectului (KDE sau terminal) şi a limbajului în care acesta este implementat (C sau C++). De asemenea, codul corespunzător interfeţei grafice a programului poate fi generat în mod automat cu ajutorul unui vrăjitor vizual de editare.

Ecranul `KDevelop` este alcătuit din trei zone:

1. Browser-ul proiectului, conţinând mai multe tab-uri: arborele de clase, arborele de fişiere al proiectului, documentaţii, fereastră de inspectare a variabilelor.
2. Editorul de texte sau mini-navigatorul Web pentru vizualizarea de documentaţii, după caz.
3. Zona rezervată mesajelor, conţinând mai multe tab-uri: mesaje generate de `KDevelop`, `stdout`, `stderr`, consolă (shell), mesaje de depanare etc.

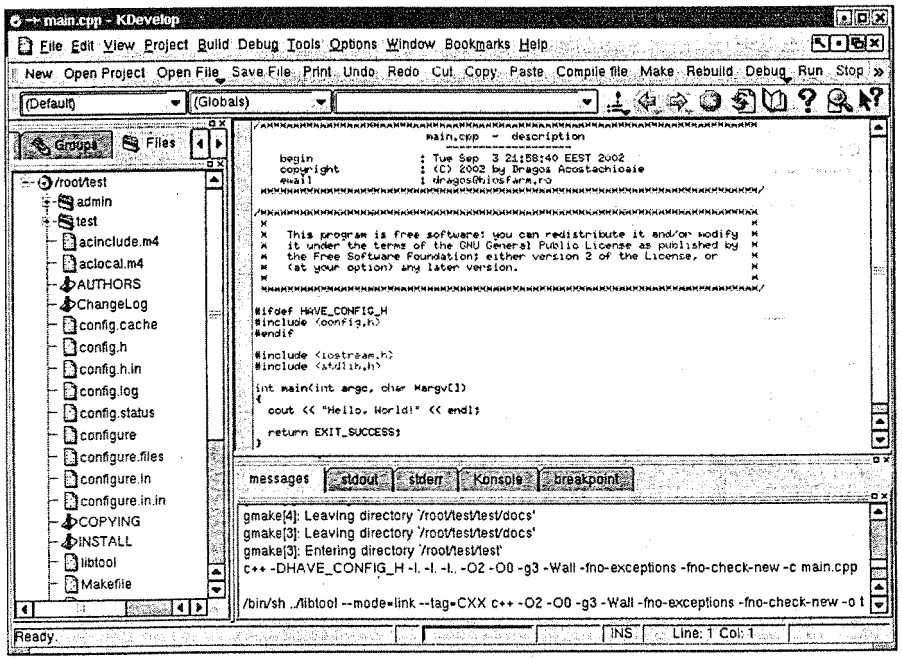


Figura 4.2. Programul KDevelop

feof	stdio.h	Verifică dacă s-a ajuns la sfârşitul fişierului
ferror	stdio.h	Verifică indicatorul de eroare
fflush	stdio.h	Scrie pe disc datele nesalvate
fgetc	stdio.h	Citeşte un caracter dintr-un fişier
fgets	stdio.h	Citeşte un şir de caractere dintr-un fişier
floor	math.h	Rotunjeşte în minus
fmod	math.h	Calculează restul împărţirii
fopen	stdio.h	Deschide un fişier
fprintf	stdio.h	Scrie date formatate într-un fişier
fputc	stdio.h	Scrie un caracter într-un fişier
fputs	stdio.h	Scrie un şir de caractere într-un fişier
fread	stdio.h	Citeşte informaţii dintr-un fişier
free	stdlib.h	Eliberează spaţiul de memorie alocat pentru un pointer
freopen	stdio.h	Redeschide un fişier
frexp	math.h	Converteşte un număr în parte fracţională şi parte întreagă
fscanf	stdio.h	Citeşte date formatate dintr-un fişier
fseek	stdio.h	Setează poziţia pointerului într-un fişier
ftell	stdio.h	Returnează poziţia cursorului într-un fişier
fwrite	stdio.h	Scrie informaţii într-un fişier
getc	stdio.h	Citeşte un caracter dintr-un fişier
getchar	stdio.h	Returnează un caracter citit de la intrarea standard
getenv	stdlib.h	Citeşte conţinutul unei variabile de mediu
gets	stdio.h	Citeşte un şir de caractere de la intrarea standard
gmtime	time.h	Transformă o valoare de timp în structura corespunzătoare
isalnum	ctype.h	Verifică dacă un caracter este alfanumeric
isalpha	ctype.h	Verifică dacă un caracter este literă
iscntrl	ctype.h	Verifică dacă un caracter este caracter de control
isdigit	ctype.h	Verifică dacă un caracter este cifră
isgraph	ctype.h	Verifică dacă un caracter este tipăribil exceptând spaţiu
islower	ctype.h	Verifică dacă un caracter este literă mică

isprint	ctype.h	Verifică dacă un caracter este tipăribil, inclusiv spaţiu
ispunct	ctype.h	Verifică dacă un caracter este tipăribil, exceptând spaţiu sau caractere alfanumerice
isspace	ctype.h	Verifică dacă un caracter este caracter de spaţiere
isupper	ctype.h	Verifică dacă un caracter este literă mare
isxdigit	ctype.h	Verifică dacă un caracter este cifră hexazecimală
ldexp	math.h	Multiplică un număr cu o putere a lui 2
localtime	time.h	Transformă o valoare de timp în structura corespunzătoare
log	math.h	Calculează logaritmul natural
log10	math.h	Calculează logaritmul natural în baza 10
longjmp	setjmp.h	Sare la o adresă în afara programului curent
malloc	stdlib.h	Alocă spaţiu de memorie pentru un pointer
mktime	time.h	Transformă o structură de timp în valoarea corespunzătoare
modf	math.h	Extrage părţile întregi şi fracţionale
perror	stdio.h	Trimite un mesaj de eroare la ieşirea de eroare standard
pow	math.h	Ridică un număr la putere oarecare
printf	stdio.h	Scrie date formatate la ieşirea standard
putc	stdio.h	Scrie un caracter într-un fişier
putchar	stdio.h	Scrie un caracter la ieşirea standard
puts	stdio.h	Scrie un şir de caractere la ieşirea standard
qsort	stdlib.h	Sorteză un tablou de elemente
rand	stdlib.h	Returnează un număr aleatoriu
realloc	stdlib.h	Modifică dimensiunea spaţiului de memorie alocat pentru un pointer
remove	stdio.h	Şterge un fişier
rename	stdio.h	Redenumeşte un fişier
rewind	stdio.h	Poziţionează pointerul de fişier la începutul acestuia
scanf	stdio.h	Citeşte date formatate de la intrarea standard
setbuf	stdio.h	Ataşează un buffer pentru fişier
setjmp	setjmp.h	Salvează mediul curent

setlocale	locale.h	Stabileşte localizarea de ţară
sin	math.h	Calculează sinusul
sinh	math.h	Calculează sinusul hiperbolic
sprintf	stdio.h	Scrie date formatate într-un şir
sqrt	math.h	Calculează radicalul
srand	stdlib.h	Setează „sămânţa" pentru generatorul de numere aleatorii
sscanf	stdio.h	Citeşte date formatate dintr-un şir
strcat	string.h	Concatenează două şiruri
strchr	string.h	Caută un caracter într-un şir
strcmp	string.h	Compară două şiruri
strcpy	string.h	Copie un şir într-un altul
strcspn	string.h	Returnează numărul de caractere dintr-un şir care este inclus într-un alt şir
strftime	time.h	Scrie echivalentul unui moment de timp într-un şir
strlen	string.h	Returnează lungimea unui şir
strncat	string.h	Concatenează două şiruri, cu o lungime maximă specificată
strncmp	string.h	Compară un număr de caractere din două şiruri
strncpy	string.h	Copie un număr de caractere dintr-un şir într-altul
strpbrk	string.h	Caută un şir de caractere într-un alt şir
strrchr	string.h	Caută un caracter într-un şir
strspn	string.h	Returnează numărul de caractere dintr-un şir care este inclus într-un alt şir
strstr	string.h	Caută un şir de caractere într-un alt şir
strtok	string.h	Caută un şablon într-un şir
tan	math.h	Calculează tangenta
tanh	math.h	Calculează tangenta hiperbolică
time	time.h	Returnează timpul curent
tmpfile	stdio.h	Deschide un fişier temporar
tmpnam	stdio.h	Returnează un nume unic de fişier temporar
tolower	ctype.h	Converteşte un caracter în literă mică
toupper	ctype.h	Converteşte un caracter în literă mare
ungetc	stdio.h	Pune un caracter înapoi într-un fişier

BIB. CENTR. UNIV. „MIHAIL KOGĂLNICEANU" IAŞI INFORMATICĂ

Anexa 2

FUNCŢIILE STANDARD POSIX

Nume funcţie	Fişier-antet	Descriere
access	unistd.h	Verifică dacă identificatorul de utilizator sub care se execută procesul curent are dreptul de acces specificat asupra fişierului furnizat ca parametru
alarm	unistd.h	Programează trimiterea semnalului SIGALRM după un interval de timp
cfgetispeed	termios.h	Returnează viteza de transfer pentru intrare (*baud rate*) a terminalului
cfgetospeed	termios.h	Returnează viteza de transfer pentru ieşiere a terminalului
cfsetispeed	termios.h	Stabileşte viteza de transfer pentru intrare
cfsetospeed	termios.h	Stabileşte viteza de transfer pentru ieşire
chdir	unistd.h	Schimbă directorul curent
chmod	sys/types.h, sys/stat.h	Modifică drepturile de acces ale unui fişier
close	unistd.h	Închide un descriptor de fişier
closedir	sys/types.h, dirent.h	Închide un descriptor de director
creat	sys/types.h, sys/stat.h, fcntl.h	Creează şi deschide pentru scriere un fişier
ctermid	stdio.h, unistd.h	Returnează numele terminalului curent
dup	unistd.h	Realizează o copie a unui descriptor de fişier
dup2	unistd.h	Închide un descriptor de fişier şi realizează o copie a sa
execl, execle, execlp, execv, execve, execvp	unistd.h	Apelează un fişier executabil, suprascriind procesul aflat în execuţie
_exit	unistd.h	Încheie execuţia procesului curent şi închide toate fişierele deschise

fcntl	sys/types.h, unistd.h, fcntl.h	Controlează sau returnează atributele unui fişier deschis
fdopen	stdio.h	Deschide un flux utilizând un descriptor de fişier deja deschis
fileno	stdio.h	Returnează descriptorul de fişier asociat unui flux
fork	sys/types.h, unistd.h	Creează un proces copil, care este o copie a procesului curent
fpathconf	unistd.h	Returnează valoarea unei variabile pentru un anumit descriptor de fişier
fstat	sys/types.h, sys/stat.h	Obţine starea curentă a unui fişier deschis
getcwd	unistd.h	Returnează calea absolută pentru directorul curent
getegid	sys/types.h, unistd.h	Returnează identificatorul de grup efectiv al procesului curent
getenv	stdlib.h, unistd.h	Returnează valoare unei anumite variabile de mediu
geteuid	sys/types.h, unistd.h	Returnează identificatorul de utilizator efectiv al procesului curent
getgid	sys/types.h, unistd.h	Returnează identificatorul de grup real al procesului curent
getgrgid	grp.h	Caută un identificator de grup în baza de date de grupuri (`/etc/group`)
getgrnam	grp.h	Caută un nume de grup în baza de date de grupuri
getgroups	sys/types.h, unistd.h	Obţine grupurile suplimentare asociate cu procesul curent
getlogin	unistd.h	Returnează numele utilizatorului curent
getpgrp	sys/types.h, unistd.h	Returnează identificatorul de grup al procesului curent
getpid	sys/types.h, unistd.h	Returnează identificatorul procesului curent
getppid	sys/types.h, unistd.h	Returnează identificatorul procesului părinte
getpwnam	pwd.h	Caută un nume de utilizator în baza de date de utilizatori (`/etc/passwd`)
getpwuid	pwd.h	Caută un identificator de utilizator în baza de date de utilizatori

getuid	sys/types.h, unistd.h	Returnează identificatorul de utilizator real al procesului curent
isatty	unistd.h	Verifică dacă un descriptor de fişier are asociat un terminal
kill	sys/types.h, signal.h	Trimite un semnal unui proces specificat
link	unistd.h	Creează o legătură simbolică
lseek	unistd.h, sys/types.h	Poziţionează pointerul curent în cadrul unui descriptor de fişier
mkdir	sys/types.h, sys/stat.h	Creează un director
mkfifo	sys/types.h, sys/stat.h	Creează un FIFO
open	sys/types.h, sys/stat.h, fcntl.h	Deschide un fişier
opendir	sys/types.h, dirent.h	Deschide un director
pathconf	unistd.h	Returnează valoarea unei variabile pentru o anumită cale
pause	unistd.h	Blochează procesul până la recepţionarea unui semnal
pipe	unistd.h	Creează un *pipe*
read	unistd.h	Citeşte informaţii dintr-un fişier
readdir	sys/types.h, dirent.h	Citeşte următoarea intrare din director
rename	unistd.h	Redenumeşte un fişier
rewinddir	sys/types.h, dirent.h	Reciteşte conţinutul unui director
rmdir	unistd.h	Şterge un director
setgid	sys/types.h, unistd.h	Setează grupul sub care este executat procesul
setpgid	sys/types.h, unistd.h	Setează grupul pentru un proces specificat
setsid	sys/types.h, unistd.h	Modifică sesiunea din care face parte procesul într-o nouă sesiune
setuid	sys/types.h, unistd.h	Modifică identificatorul de utilizator sub care se execută procesul
sigaction	signal.h	Asignează tabelul de tratamente ale semnalelor

sigaddset	signal.h	Adaugă un semnal la o tabelă de tratamente ale semnalelor
sigdelset	signal.h	Şterge un semnal dintr-o tabelă de tratamente ale semnalelor
sigemptyset	signal.h	Şterge toate semnalele dintr-o tabelă de tratamente ale semnalelor
sigfillset	signal.h	Adaugă toate semnalele cunoscute într-o tabelă de tratamente ale semnalelor
sigismember	signal.h	Verifică dacă un semnal este membru al unei tabele de tratamente ale semnalelor
siglongjmp	setjmp.h	Execută salt la un context salvat anterior cu `sigsetjmp`
sigpending	signal.h	Modifică tabelul de tratamente ale semnalelor
sigsetjmp	setjmp.h	Salvează contextul curent
sigsuspend	signal.h	Modifică masca de semnale şi blochează procesul până la recepţionarea unui semnal
sleep	unistd.h	Blochează procesul pentru un interval de timp
stat	sys/types.h, sys/stat.h	Obţine informaţii de stare privind un anumit fişier
sysconf	unistd.h	Returnează valoarea unei variabile sistem
tcdrain	termios.h	Blochează procesul până când toate informaţiile scrise într-un descriptor de fişier au fost transmise
tcflow	termios.h	Opreşte şi reporneşte operaţiunile de intrare/ieşire cu un anumit descriptor de fişier
tcflush	termios.h	Termină operaţiunile de intrare/ieşire rămase de efectuat cu un anumit descriptor de fişier
tcgetattr	termios.h	Obţine valorile atributelor asociate unui terminal
tcgetpgrp	termios.h	Returnează identificatorul de grup al procesului asociat unui terminal
tcsendbreak	termios.h	Trimite către un terminal un şir de octeţi având valoarea zero, pentru un anumit timp
tcsetattr	termios.h	Stabileşte valorile atributelor asociate unui terminal
tcsetpgrp	termios.h	Stabileşte identificatorul de grup al procesului asociat unui terminal

time	time.h	Returnează timpul curent (exprimat în secunde trecute de la începutul epocii, adică 1 ianuarie 1970)
times	sys/times.h	Obţine timpul utilizator şi timpul sistem asociate procesului, precum şi cele asociate proceselor copil
ttyname	unistd.h	Returnează numele unui terminal
tzset	time.h	Stabileşte informaţiile de conversie a timpului
umask	sys/types.h, sys/stat.h	Stabileşte masca pentru drepturile de acces la crearea fişierelor
uname	sys/utsname.h	Obţine informaţii de identificare asupra sistemului
unlink	stdio.h	Şterge un fişier
utime	sys/types.h, utime.h	Stabileşte timpul de acces, respectiv de modificare a unui fişier
wait	sys/types.h, sys/wait.h	Blochează procesul curent până când se încheie un proces copil
waitpid	sys/types.h, sys/wait.h	Obţine informaţii despre un anumit proces copil care şi-a încheiat execuţia sau a fost oprit
write	unistd.h	Scrie informaţii într-un fişier

Anexa 3

LICENȚA PUBLICĂ GNU

Aceasta este o traducere neoficială a Licenței Publice Generale GNU în limba română. Nu a fost publicată de Free Software Foundation și nu specifică termenii legali de distribuire a programelor care folosesc GNU GPL – numai textul original în limba engleză al GNU GPL face acest lucru. Totuși, sperăm că această traducere să ajute vorbitorii de limba română să înțeleagă mai bine GNU GPL. Textul traducerii a fost preluat de la adresa: `http://www.roedu.net/gplro.html`

Versiunea originală a acestei licențe poate fi găsită la adresa: `http://www.gnu.org/licenses/gpl.html`.

Licența publică generală GNU

Versiunea 2, iunie 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Este permisă copierea acestui document, dar este interzisă modificarea lui.

Prefață

Licențele majorității programelor sunt concepute pentru a vă priva de libertatea de a modifica și distribui programele respective. În contrast, intenția Licenței Publice Generale GNU este de a vă garanta libertatea de a distribui și modifica programele libere și de a se asigura că programele sunt libere pentru toți utilizatorii. Această Licență Publică Generală se aplică majorității programelor aparținând Free Software Foundation precum și tuturor celorlalte programe ai căror autori decid să o folosească. Alte programe aparținând Free Software Foundation sunt puse sub Licența Publică Generală GNU pentru Biblioteci. Această Licență poate fi de asemenea folosită pentru programele dumneavoastră.

Libertatea programelor nu implică neapărat absența costului. Licențele noastre sunt concepute să vă garanteze libertatea de a distribui copii ale programelor libere (și de a oferi

acest serviciu contra cost, dacă doriți), de a obține codul-sursă, de a schimba programul sau a folosi porțiuni din el în noi programe libere, și de a ști că puteți face toate aceste lucruri.

Pentru a vă proteja drepturile, trebuie să impunem restricții împotriva oricui ar încerca să vă conteste aceste drepturi sau să vă ceară să renunțați la ele. Aceste restricții implică anumite responsabilități pentru dumneavoastră dacă distribuiți copii ale programelor, sau dacă le modificați.

De exemplu, dacă distribuiți copii ale unui program, indiferent dacă o faceți gratuit sau contra unei sume de bani, trebuie să dați beneficiarilor toate drepturile pe care le aveți dumneavoastră. Trebuie să vă asigurați că ei primesc, sau pot primi, codul-sursă al programului. În plus, trebuie să le arătați care sunt termenii în care primesc programul, pentru ca ei să știe care le sunt drepturile.

Drepturile dumneavoastră sunt protejate în două etape: (1) prin stabilirea drepturilor de autor pentru program, și (2) prin această Licență care vă dă dreptul legal de a copia, distribui și/sau modifica programul.

De asemenea, pentru propria noastră protecție cât și pentru cea a autorilor, vrem să ne asigurăm că toată lumea înțelege că nu există nici un fel de garanție pentru acest program liber. Dacă programul este modificat de altcineva și distribuit mai departe, vrem ca beneficiarii programului să știe că ceea ce au nu este originalul, în așa fel încât nici o problemă introdusă de altcineva nu va avea un efect negativ asupra reputației autorilor inițiali.

Orice program liber este în mod constant amenințat de patentele software. Noi vrem să evităm pericolul ca cei ce redistribuie programe libere să obțină patente, practic transformând programul într-unul aflat sub controlul total al persoanei sau instituției ce deține patentul (engl. *proprietary*). Pentru a preveni această situație, facem clară poziția noastră conform căreia orice patent trebuie acordat fie în așa fel încât să poată fi folosit gratuit și fără restricții de oricine, fie deloc.

Termenii și condițiile exacte de copiere, distribuire și modificare sunt specificate în următoarele paragrafe.

Licența publică generală GNU

Termeni și condiții pentru copiere, distribuire și modificare

0. Această Licență se aplică oricărui program sau proiect ce conține o mențiune a deținătorului drepturilor de autor spunând că poate fi distribuit în termenii acestei Licențe Publice Generale. Prin „Program" vom înțelege orice asemenea program sau proiect, iar prin „proiect bazat pe Program" vom înțelege fie programul fie orice alt proiect derivat din Program conform cu legea drepturilor de autor: un proiect ce conține Programul sau porțiuni din el, fie în forma originală, fie modificată și/sau tradusă în altă limbă (în restul acestui document traducerile vor fi incluse fără restricții în termenul „modificare"). Fiecare persoană autorizată de această Licență va fi desemnată prin termenul „dumneavoastră".

Activităţile care nu sunt de copiere, distribuire şi modificare se află în afara scopului acestei Licenţe. Activitatea de executare a programului nu este restricţionată, iar rezultatul programului (engl. *output*) este acoperit de licenţă doar în cazul în care conţinutul său constituie un proiect bazat pe Program (independent de faptul că a fost obţinut prin rularea Programului). În ce măsură acest lucru este adevărat depinde de natura Programului.

1. Puteţi copia şi distribui copii nemodificate ale codului-sursă al Programului în forma în care îl primiţi, prin orice mediu, cu condiţia să specificaţi vizibil pe fiecare copie autorul şi lipsa oricărei garanţii, să păstraţi intacte toate notele referitoare la această Licenţă şi la absenţa oricărei garanţii şi să distribuiţi o copie a acestei Licenţe cu fiecare copie a Programului.

Puteţi pretinde o retribuţie financiară pentru actul fizic de transfer al unei copii, şi puteţi oferi garanţie contra cost.

2. Puteţi efectua modificări asupra copiilor Programului (sau asupra oricăror porţiuni ale sale), creând astfel un „proiect bazat pe Program". Copierea şi distribuirea unor asemenea modificări sau proiecte se pot face conform termenilor secţiunii precedente (1), doar dacă toate condiţiile următoarele sunt îndeplinite:

a. toate fişierele modificate trebuie să conţină note foarte vizibile menţionând faptul că dumneavoastră le-aţi modificat, precum şi data fiecărei modificări;
b. orice proiect pe care îl distribuiţi sau publicaţi, care în întregime sau în parte conţine sau este derivat din Program (sau orice parte a acestuia), trebuie să poată fi folosit de oricine, gratuit şi în întregime, în termenii acestei Licenţe;
c. dacă programul modificat citeşte comenzi în mod interactiv, trebuie să îl modificaţi în aşa fel încât atunci când este pornit în mod interactiv să afişeze un mesaj referitor la drepturile de autor precum şi o notă menţionând lipsa oricărei garanţii (sau să menţioneze faptul că dumneavoastră oferiţi o garanţie). De asemenea trebuie specificat faptul că utilizatorii pot redistribui programul în aceste condiţii precum şi o explicaţie a modalităţii în care poate fi obţinut textul acestei Licenţe (excepţie: dacă Programul este interactiv dar nu afişează în mod normal un asemenea mesaj, nu este necesar ca proiectul bazat pe Program să afişeze un mesaj).

Aceste cerinţe se aplică Programului modificat în întregime. Dacă pot fi identificate secţiuni ale proiectului care nu sunt derivate din Program, şi pot fi considerate de sine stătătoare, atunci această Licenţă şi termenii săi nu se aplică acelor secţiuni când sunt distribuite ca proiecte separate. Când distribuiţi aceleaşi secţiuni ca parte a unui întreg care este un proiect bazat pe Program, distribuirea întregului proiect trebuie să fie făcută în acord cu termenii acestei Licenţe, ale cărei permisiuni pentru alte licenţe se extind asupra întregului, şi deci asupra fiecărei secţiuni în parte, indiferent de autor.

Astfel, nu este în intenţia acestei secţiuni să pretindă drepturi sau să conteste drepturile dumneavoastră asupra unui proiect efectuat în întregime de dumneavoastră. Intenţia este de a exercita dreptul de a controla distribuţia proiectelor derivate sau colective bazate pe Program.

În plus, pura agregare (pe un mediu de stocare sau distribuţie) cu Programul (sau cu un proiect bazat pe Program) al unui alt proiect care nu este bazat pe Program nu aduce acel proiect sub incidenţa acestei Licenţe.

3. Puteţi copia şi distribui Programul (sau un proiect bazat pe el, conform Secţiunii 2) în format obiect sau executabil conform termenilor Secţiunilor 1 şi 2 de mai sus, cu condiţia să îndepliniţi una dintre condiţiile de mai jos:

a. să îl oferiţi însoţit de codul-sursă corespunzător, în format citibil de către maşină, care trebuie să fie distribuit în termenii Secţiunilor 1 şi 2 de mai sus pe un mediu de distribuţie uzual transportului de software; sau
b. să îl oferiţi însoţit de o ofertă scrisă (valabilă pentru cel puţin trei ani, pentru o taxă care să nu depăşească costul fizic al efectuării distribuţiei sursei), de a oferi o copie completă, în format citibil de către maşină, a codului-sursă, distribuit în termenii Secţiunilor 1 şi 2 de mai sus, pe un mediu de distribuţie uzual transportului de software; sau
c. să îl oferiţi însoţit de informaţia pe care aţi primit-o referitoare la oferta de a distribui codul-sursă corespunzător. (Această alternativă este permisă numai pentru distribuiri necomerciale şi doar dacă aţi primit programul în format obiect sau executabil împreună cu această ofertă, în conformitate cu Subsecţiunea b de mai sus.)

Codul-sursă al unui proiect este forma preferată în care se fac modificări asupra proiectului. Pentru un proiect executabil, codul-sursă complet înseamnă codul-sursă al tuturor modulelor pe care le conţine, împreună cu toate fişierele asociate conţinând definiţii ale interfeţelor şi scripturile folosite pentru a controla compilarea şi instalarea executabilului. Cu toate acestea, ca o excepţie, nu este obligatorie distribuirea împreună cu codul-sursă a acelor componente care sunt în mod normal distribuite (în format-sursă sau binar) cu componentele majore (compilator, nucleu etc.) ale sistemului de operare sub care rulează executabilul, exceptând situaţia în care acea componentă acompaniază executabilul.

Dacă distribuţia executabilului sau codului-obiect este făcută prin oferirea permisiunii de copiere dintr-un loc dedicat, atunci oferirea permisiunii de copiere a codului-sursă din acelaşi loc este considerată distribuire a codului-sursă, chiar dacă beneficiarul nu este obligat să copieze codul-sursă împreună cu codul-obiect.

4. Nu puteţi copia, modifica, sub-autoriza sau distribui Programul decât aşa cum este prevăzut în această Licenţă. Orice încercare de a copia, modifica, sub-autoriza sau distribui Programul în alţi termeni va duce la anularea drepturilor ce vă revin conform acestei Licenţe. Cu toate acestea, nu vor fi anulate drepturile celor ce au primit copii sau drepturi de la dumneavoastră conform cu această Licenţă, atât timp cât rămân în conformitate cu ea.

5. Nu sunteţi obligat să acceptaţi această Licenţă, deoarece nu aţi semnat-o. Totuşi, numai această Licenţă vă permite să modificaţi Programul sau proiectele derivate din el. Aceste acţiuni sunt interzise prin lege dacă nu acceptaţi această Licenţă. În consecinţă, prin modificarea sau distribuirea Programului (sau a oricărui proiect bazat pe Program), indicaţi în mod implicit acceptarea acestei Licenţe şi a tuturor termenilor şi condiţiilor de copiere, distribuire sau modificare a Programului sau proiectelor bazate pe el.

6. De fiecare dată când redistribuiţi Programul (sau orice proiect bazat pe Program), beneficiarul primeşte o licenţă de la licenţiatorul original care îi permite să copieze, distribuie sau modifice Programul în aceiaşi termeni şi condiţii. Nu puteţi impune nici o restricţie adiţională asupra exercitării drepturilor pe care destinatarul le primeşte prin această Licenţă. Nu sunteţi responsabil cu impunerea respectării acestei Licenţe de către o terţă parte.

7. În cazul în care, ca o consecinţă a unei decizii judecătoreşti, sau pretinsă încălcare a unui patent, sau pentru orice altă cauză (nu neapărat limitată la chestiuni legate de patente), vi se impun condiţii (prin hotărâre judecătorească, înţelegere sau alte mijloace) care contravin condiţiilor acestei Licenţe, acest lucru nu vă permite nerespectarea condiţiilor Licenţei. Dacă nu puteţi face în aşa fel încât să satisfaceţi simultan obligaţiile din această

Licenţă şi alte obligaţii pertinente, atunci, ca o consecinţă, vă este interzisă distribuirea Programului. De exemplu, dacă o autorizaţie de folosire a unui patent nu vă permite redistribuirea gratuită a Programului de către oricine îl primeşte de la dumneavoastră, direct sau indirect, atunci singurul mod în care puteţi satisface simultan aceste condiţii şi Licenţa de faţă este să nu distribuiţi Programul în nici un fel.

Dacă vreo porţiune a acestei secţiuni este invalidată sau de neaplicat în anumite circumstanţe, restul secţiunii continuă să se aplice, iar secţiunea în întregime se aplică în toate celelalte circumstanţe.

Nu este în intenţia acestei secţiuni să vă determine să încălcaţi vreun patent sau alte pretenţii de drepturi de proprietate, sau să contestaţi valabilitatea oricăror asemenea pretenţii; această secţiune are drept unic scop protejarea integrităţii sistemului de distribuire de programe libere, care este implementat prin licenţe publice. Multe persoane au contribuit generos la spectrul larg de programe distribuite prin acest sistem, bazându-se pe aplicarea sa consistentă; este la latitudinea autorului/donatorului să decidă dacă este dispus să distribuie programe prin orice alt sistem, şi o persoană autorizată să folosească acele programe nu poate impune acea decizie.

Intenţia acestei secţiuni este de a clarifica ceea ce este considerat a fi o consecinţă a restului acestei Licenţe.

8. Dacă distribuţia şi/sau folosirea Programului sunt restricţionate în anumite ţări din cauza patentelor sau din cauza unor interfeţe aflate sub incidenţa unor drepturi de autor restrictive, deţinătorul drepturilor de autor ce plasează Programul sub această Licenţă poate adăuga o limitare geografică a distribuirii ce exclude acele ţări, în aşa fel încât distribuirea este permisă doar în (sau între) ţările care nu sunt excluse. Într-un asemenea caz, Licenţa incorporează această limitare ca şi cum ar fi scrisă în corpul acestei Licenţe.

9. Free Software Foundation poate publica din când în când noi versiuni (sau versiuni revăzute) ale Licenţei Publice Generale. Asemenea versiuni noi vor fi similare în spirit versiunii prezente, dar pot diferi în anumite detalii, pentru a adresa noi probleme sau situaţii.

Fiecărei versiuni îi este asociat un număr unic. Dacă programul specifică faptul că i se aplică un număr de versiune al acestei Licenţe şi „orice versiune ulterioară", aveţi opţiunea de a urma termenii şi condiţiile acelei versiuni sau ale oricărei versiuni ulterioare publicate de Free Software Foundation. Dacă Programul nu specifică un număr de versiune, puteţi alege orice versiune publicată vreodată de Free Software Foundation.

10. Dacă doriţi să incorporaţi părţi ale Programului în alte programe libere ale căror condiţii de distribuţie sunt diferite, cereţi permisiunea autorului. Pentru programe ale căror drepturi de autor aparţin Free Software Foundation, cereţi permisiunea de la Free Software Foundation; uneori facem excepţii pentru aceasta. Decizia noastră va fi ghidată de cele două scopuri de a prezerva statutul liber al tuturor proiectelor derivate din programele noastre libere şi de a promova distribuirea şi refolosirea programelor în general.

Nici o garanţie

11. DEOARECE PROGRAMUL ESTE OFERIT SUB O LICENŢĂ CE NU IMPLICĂ NICI UN COST, NU EXISTĂ NICI O GARANŢIE PENTRU PROGRAM, ÎN MĂSURA PERMISĂ DE LEGILE CE SE APLICĂ. EXCEPTÂND SITUAŢIILE UNDE ESTE

SPECIFICAT ALTFEL ÎN SCRIS, DEŢINĂTORII DREPTURILOR DE AUTOR ŞI/SAU ALTE PĂRŢI IMPLICATE OFERĂ PROGRAMUL „ÎN FORMA EXISTENTĂ" FĂRĂ NICI O GARANŢIE, EXPLICITĂ SAU IMPLICITĂ, INCLUZÂND, DAR FĂRĂ A FI LIMITATĂ LA GARANŢII IMPLICITE DE VANDABILITATE ŞI CONFORMITATE UNUI ANUMIT SCOP. VĂ ASUMAŢI ÎN ÎNTREGIME RISCUL ÎN CEEA CE PRIVEŞTE CALITATEA ŞI PERFORMANŢA ACESTUI PROGRAM. ÎN CAZUL ÎN CARE PROGRAMUL SE DOVEDEŞTE A FI DEFECT, VĂ ASUMAŢI ÎN ÎNTREGIME COSTUL TUTUROR SERVICIILOR, REPARAŢIILOR ŞI CORECŢIILOR NECESARE.

12. ÎN NICI O SITUAŢIE, EXCEPTÂND CAZURILE ÎN CARE ESTE CERUT DE LEGEA APLICABILĂ SAU CA REZULTAT AL UNEI ÎNŢELEGERI SCRISE, UN DEŢINĂTOR AL DREPTURILOR DE AUTOR, SAU ORICE ALTĂ PARTE CARE POATE MODIFICA ŞI/SAU REDISTRIBUI PROGRAMUL CONFORM PERMISIUNILOR DE MAI SUS, NU VA FI FĂCUT RĂSPUNZĂTOR PENTRU PAGUBELE DUMNEAVOASTRĂ, INCLUSIV CELE GENERALE, SPECIALE, ÎNTÂMPLĂTOARE SAU REZULTANTE, APĂRUTE DIN FOLOSIREA SAU INABILITATEA DE A FOLOSI PROGRAMUL (INCLUZÂND, DAR FĂRĂ A FI LIMITAT LA PIERDEREA SAU DETERIORAREA DATELOR, SAU PIERDERILE SUFERITE DE DUMNEAVOASTRĂ SAU TERŢE PERSOANE, SAU O INCAPACITATE A PROGRAMULUI DE A INTEROPERA CU ALTE PROGRAME), CHIAR DACĂ DEŢINĂTORUL SAU TERŢA PARTE AU FOST PREVENIŢI ASUPRA POSIBILITĂŢII UNOR ASEMENEA PAGUBE.

Sfârşitul termenilor şi condiţiilor

Cum să *Aplicaţi Aceşti Termeni Noilor Dumneavoastră Programe*?

Dacă dezvoltaţi un nou program, şi doriţi să fie de cea mai mare utilitate publicului, cea mai bună metodă de a realiza acest lucru este să-l faceţi liber, în aşa fel încât oricine să-l poată redistribui şi modifica în aceşti termeni.

Pentru a face acest lucru, ataşaţi următoarea notă programului. Cel mai sigur este să o ataşaţi începutului fiecărui fişier-sursă pentru a transmite excluderea garanţiei; de asemenea, fiecare fişier ar trebui să conţină cel puţin linia conţinând drepturile de autor şi o referire la locul unde poate fi găsită întreaga notă.

<o linie cu numele programului şi o scurtă prezentare a ceea ce face.>
Copyright (C) 19yy <numele autorului>

Acest program este liber; îl puteţi redistribui şi/sau modifica în conformitate cu termenii Licenţei Publice Generale GNU, aşa cum este publicată de Free Software Foundation; fie versiunea 2 a Licenţei, fie (la latitudinea dumneavoastră) orice versiune ulterioară.

Acest program este distribuit cu speranţa că va fi util, dar FĂRĂ NICI O GARANŢIE, fără garanţie implicită de vandabilitate şi conformitate unui anumit scop. Citiţi Licenţa Publică Generală GNU pentru detalii.

Ar trebui să fi primit o copie a Licenţei Publice Generale GNU împreună cu acest program; dacă nu, scrieţi Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

De asemenea, specificaţi modalităţile în care puteţi fi contactat prin poştă normală şi electronică.

Dacă programul este interactiv, modificaţi-l în aşa fel încât să genereze o scurtă notă atunci când porneşte în mod interactiv:

Gnomovision versiunea 69, Copyright (C) 19yy numele autorului Gnomovision nu oferă ABSOLUT NICI O GARANŢIE; pentru detalii tastaţi *„arată g"*. Acest program este liber, şi sunteţi bine veniţi să-l redistribuiţi în anumite condiţii; tastaţi *„arată c"* pentru detalii.

Comenzile ipotetice *„arată g"* şi *„arată c"* ar trebui să arate porţiunile corespunzătoare din Licenţa Publică Generală. Bineînţeles, comenzile pe care le veţi folosi pot fi denumite altfel decât *„arată g"* şi *„arată c"*; ele pot fi plasate pe meniuri, acţiuni ale mouse-ului – orice este potrivit pentru programul dumneavoastră.

De asemenea, ar trebui să obţineţi un document semnat de instituţia pentru care lucraţi (dacă lucraţi ca programator) sau de şcoală/universitate, prin care aceasta renunţă la drepturile de autor pentru programul dumneavoastră. Urmează un exemplu; modificaţi numele:

Yoyodyne, Inc., prin aceasta renunţă la toate drepturile de autor asupra programului „Gnomovision" (care face cu ochiul compilatoarelor) scris de James Hacker.

<semnatura lui Ty Coon>, 1 April 1989 Ty Coon, President of Vice

Această Licenţă Publică Generală nu permite incorporarea programului dumneavoastră în programe aflate sub controlul restrictiv al instituţiilor (engl. *proprietary programs*). Dacă programul dumneavoastră este o subrutină a unei biblioteci, puteţi considera mai util să permiteţi legarea la bibliotecă a programelor aflate sub controlul restrictiv al instituţiilor. Dacă aceasta este ceea ce doriţi, folosiţi Licenţa Publică Generală GNU pentru Biblioteci în locul acestei Licenţe.

BIBLIOGRAFIE

1. Satir Gregory, Brown Doug, *C++: The Core Language*, O'Reilly & Associates, Sebastopol, 1995
2. Oualline Steve, *Practical C++ Programming*, O'Reilly & Associates, Sebastopol, 1997
3. Bruce Eckel, *Thinking in C++*, Second Edition, Prentice Hall, New Jersey, 2000
4. Danny Kalev, *ANSI/ISO C++ Professional Programmer's Handbook*, Macmillan Computer Publishing, 1999
5. Coplien O. James, *Advanced Programming Styles and Idioms*, Addison-Wesley Publishing, Massachusetts, 1992
6. Booch Grady, *Object-Oriented Analysis and Design – With applications*, Benjamin/Cummings Publishing, Redwood City, 1994
7. Booch Grady, *Object Solutions – Managing the object-oriented project*, Addison-Wesley Publishing, Menlo Park, 1996
8. Salomie Ioan, *Tehnici Orientate pe Obiecte*, microInformatica, Cluj-Napoca, 1995
9. Fred Zlotnick, *The POSIX.1 Standard – A Programmer's Guide*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, 1991
10. ***, *Open source. Voices from the Open source Revolution*, O'Reilly & Associates, Inc., 1999: `http://www.oreilly.com/catalog/opensources/book/toc.html`

Resurse Web

1. `http://std.dkuug.dk/JTC1/SC22/WG14`: standardul ISO/IEC JTC1/SC22/WG14-C (ANSI C)
2. `ftp://ftp.research.att.com/dist/c++std/WP/CD2/`: standardul ANSI C++
3. `http://www.msoe.edu/~tritt/cpplang.html`: C++ Language Reference
4. `http://www.sgi.com/Technology/STL`: C++ STL Reference
5. `http://www.cs.umd.edu/users/cml/style/`: ghid privind stilul de programare în C şi C++
6. `http://www.doc.ic.ac.uk/lab/cplus/c++.rules`: reguli şi recomandări pentru programarea în C++
7. `http://www.programmingtutorials.com`: tutoriale de programare pentru C, C++ precum şi pentru alte limbaje
8. `http://www.gnu.org/prep/standards.html`: standardele de programare GNU
9. `http://www.snippets.org`: exemple de programe C

10. `http://www.cs.bell-labs.com/who/dmr/index.html`: situl personal al lui Dennis Ritchie
11. `http://www.research.att.com/~bs`: situl personal al lui Bjarne Stroustrup
12. `http://www.cuj.com`: C/C++ Users Journal
13. `http://www.lysator.liu.se/C/`: numeroase legături către situri referitoare la C
14. `http://www.linux.org`: situl oficial al sistemului de operare Linux
15. `http://www.gnu.org`: situl fundaţiei GNU
16. `http://www.gnu.org/software/automake`: situl Automake
17. `http://www.cvshome.org`: situl CVS
18. `http://www.gnu.org/software/ddd`: situl DDD
19. `http://docpp.sourceforge.net`: situl DOC++
20. `http://www.gnu.org/software/gcc/gcc.html`: situl GCC
21. `http://glade.gnome.org`: situl Glade
22. `http://www.kdevelop.org`: situl KDevelop
23. `http://www.splint.org`: situl SPLint
24. `http://sourceforge.net`: cel mai puternic sit dedicat dezvoltării de programe *open source*
25. `http://freshmeat.net`: cea mai mare bază de date de programe *open source*
26. `http://www.ibiblio.org`: cea mai mare bibliotecă de informaţii publice

GLOSAR DE TERMENI

Iată semnificaţiile termenilor importanţi folosiţi în această lucrare:

Asamblare

Compilare din limbaj de asamblare în cod maşină (engl. *assembly*).

ANSI

American National Standards Institute, instituţie americană care se ocupă de crearea standardelor.

Bibliotecă

O colecţie de module-obiect grupate într-un fişier-arhivă. Conţine codul unor funcţii şi definiţiile unor variabile utile (engl. *library*).

Buffer

Zonă tampon de memorie, utilizată în general pentru stocarea unor date temporare.

Bug

Eroare de proiectare, de programare etc., într-un program.

Compilare

Transformarea unui program dintr-un limbaj în cod maşină (engl. *compilation*).

Declaraţie

O construcţie care anunţă existenţa unui obiect (funcţie sau variabilă), fără însă a-l defini (construi sau aloca spaţiu de memorie) (engl. *declaration*).

Definiţie

O structură care construieşte sau alocă spaţiu de memorie pentru un obiect (engl. *definition*).

Fişier-antet

Un fişier care grupează declaraţiile obiectelor exportate de unul sau mai multe module (engl. *header file*).

Link-editare

Combinarea mai multor module-obiect într-unul singur (fişier executabil sau bibliotecă) (engl. *linking*).

HTML (HyperText Markup Language)

Lingua franca a spaţiului WWW, aplicaţie majoră a *SGML* (*Standard Generalized Markup Language*).

Maşină

Exprimare echivalentă la calculator sau sistem (engl. *host*).

Modul

O parte dintr-un program, constând dintr-un singur fişier, care grupează de obicei obiecte (variabile, funcţii etc.) folosite împreună (engl. *module*).

Platformă

Desemnează o combinaţie specifică de hardware şi software (tip sau versiune de sistem de operare – Linux, MacOS, Windows – şi/sau compilator ori limbaj de programare – GNU C/C++, Perl, Java etc.) (engl. *platform*).

Preprocesare

O fază anterioară compilării, care prelucrează textul programului-sursă C, obţinând o nouă sursă C, rezolvând macrourile din cadrul acesteia (engl. *preprocessing*).

Program executabil

Un fişier (exprimat în limbaj maşină) care poate fi direct executat de către sistemul de operare (engl. *executable file, binary*).

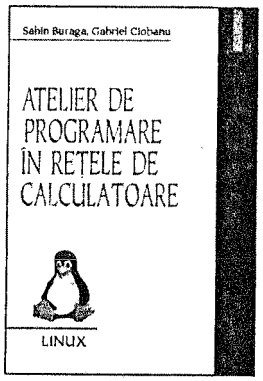
Script

Un limbaj de tip *script* este destinat să prelucreze, să automatizeze şi să integreze facilităţile oferite de un anumit sistem. Pentru sistemul de operare Linux/UNIX este disponibilă o pleiadă de interpretoare de scripturi: bash, Perl, Tcl/Tk, Python, M4 etc.

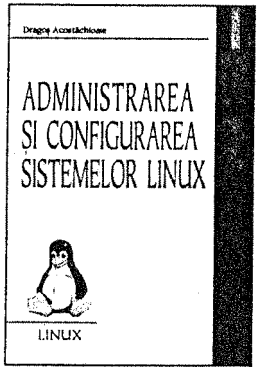
UNIX

Sistem de operare interactiv apărut în anul 1968, leagănul protocoalelor de comunicaţie în reţea şi a celor mai multe dintre conceptele teoriei şi practicii sistemelor de operare. Părinţii sistemului UNIX sunt consideraţi Ken Thompson şi Dennis Ritchie (acesta din urmă este co-inventatorul limbajului C, împreună cu Brian Kernigham). De-a lungul anilor, sistemul a suferit mutaţii importante şi s-a diversificat în diverse subtipuri (implementări) precum AIX, BSD, HP-UX, Linux, System V sau Xenix. Este utilizat de cei mai mulţi dintre furnizorii de servicii Internet şi în mediul academic. Una dintre implementările de succes este Linux, conceput de Linus Torvalds, fiind disponibil gratuit pe Internet, din 1991. Linux se află în plină dezvoltare şi este disponibil în mai multe „distribuţii" dintre care se pot enumera Red Hat, Mandrake, SuSE ori Slackware.

La Editura POLIROM
au apărut:



- Unix şi Linux
- Gestiunea fişierelor
- Procese
- Semnale
- Comunicarea între procese. *Pipe*-uri
- Duplicarea descriptărilor. Redirectări
- Interfaţa *soket*
- Modelul client/server TCP
- Modelul client/server UDP
- Multiplexarea intrărilor/ieşirilor
- RPC – Apelul procedurilor la distanţă
- Utiliyarea bibliotecii *MySQL*
- Biblioteca *ncurses*
- Mediul *Glade*



- Instalarea Linux-ului • Gestionarea pachetelor de programe
- Administrarea utilizatorilor • Sistemul X Windows • Nucleul Linux
- Legătura la reţea • Configurarea dispozitivelor hardware
- Realizarea de copii de siguranţă
- Tipărirea la imprimantă • Sisteme de baze de date • Sisteme de fişiere în reţea
- Accesarea de la distanţă a sistemului • Sistemul numelor de domenii
- Linux ca server Web • Interacţiunea cu sisteme Windows • Sistemul de poştă electronică
- Execuţia de proceduri automate • Autentificarea utilizatorilor
- Securizarea sistemului • Configurarea unui firewall



- Protocolul HTTP (HyperText Transfer Protocol)
- Limbajele de marcare şi aplicaţiile lor • Standardul CGI (Common Gateway Interface)
- Cookie-urile
- Programarea CGI în bash
- Scripturi CGI în bash
- Limbajul şi modulele Perl
- Programarea CGI în Perl • Perl şi bazele de date relaţionale
- Prelucrarea documentelor XML

www.polirom.ro

Coperta: Ionuţ Broştianu
Redactare: Andrei Dimitriu
Tehnoredactare: Irina Lăcătuşu

Bun de tipar: octombrie 2002. Apărut: 2002
Editura Polirom, B-dul Copou nr. 4 • P.O. Box 266, 6600, Iaşi
Tel. & Fax (0232) 21.41.00; (032) 21.41.11;
(0232) 21.74.40 (difuzare); E-mail: office@polirom.ro
Bucureşti, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33 • P.O. BOX 1-728, 70700
Tel.: (021) 313.89.78, E-mail: polirom@dnt.ro

Tiparul executat la SC LUMINA TIPO s.r.l.
str. Luigi Galvani nr. 20 bis, sect. 2, Bucureşti
Tel./Fax: 211.32.60, 212.29.27, E-mail: lumina-lex@fx.ro